

操作系统实用教程 螺旋方法

拉米兹·埃尔玛斯瑞 A. 吉尔·卡里克 戴维·莱文
(Ramez Elmasri) (A. Gil Carrick) (David Levine) 著
[美] 得克萨斯大学阿灵顿分校
翟高寿 译

Operating Systems
A Spiral Approach

Operating Systems
A Spiral Approach

Ramez Elmasri

A. Gil Carrick

David Levine



机械工业出版社
China Machine Press

操作系统实用教程 螺旋方法

Operating Systems A Spiral Approach

对于操作系统这门计算机专业必修课，大多数教材采用线性教学方法，以深度为导向孤立地介绍各个模块，最后整合起来理解真正的操作系统。而本书采用的螺旋方法则以广度为导向，首先给出一些基本概念，然后描述一个非常简单的操作系统，之后逐步将其演化为拥有更多功能的复杂系统。

相比之下，螺旋方法有利于学生在课程初期自然形成对操作系统各模块的认识与理解，同时不断积累信心来处理更为复杂的问题，循序渐进，从而更透彻地理解操作系统的本质。

本书特色

- 在讨论不同的操作系统时，会还原到其所在的历史背景中，介绍当时的行业状况、重要企业和个人，便于学生更好地理解操作系统的发展和演进。
- 涵盖各类便携式设备上的现代操作系统，而不限于计算机操作系统。
- 每章都配有习题，许多章节还配有实验，帮助学生巩固所学知识，在实践中强化理解。

作者简介

拉米兹·埃尔玛斯瑞 (Ramez Elmasri) 知名计算机科学家，得克萨斯大学阿灵顿分校教授。他拥有斯坦福大学计算机科学硕士和博士学位。

A. 吉尔·卡里克 (A. Gil Carrick) 曾任教于得克萨斯大学阿灵顿分校，现已退休。他是计算机科学荣誉协会Upsilon Pi Epsilon的成员。

戴维·莱文 (David Levine) 长期讲授操作系统、软件工程、网络和计算机体系结构等课程，研究兴趣包括移动计算、移动对象和分布式计算。

译者简介

翟高寿 北京交通大学计算机学院副教授，计算机科学系副主任。现主要从事操作系统、系统安全、系统软件设计等相关方向的科研和教学工作。



华章教育服务微信号

Mc
Graw
Hill
Education

www.mheducation.com



上架指导：计算机/操作系统

ISBN 978-7-111-58819-1



9 787111 588191 >

定价：99.00元

投稿热线：(010) 88379604

客服热线：(010) 88378991 88361066

购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com

网上购书：www.china-pub.com

数字阅读：www.hzmedia.com.cn

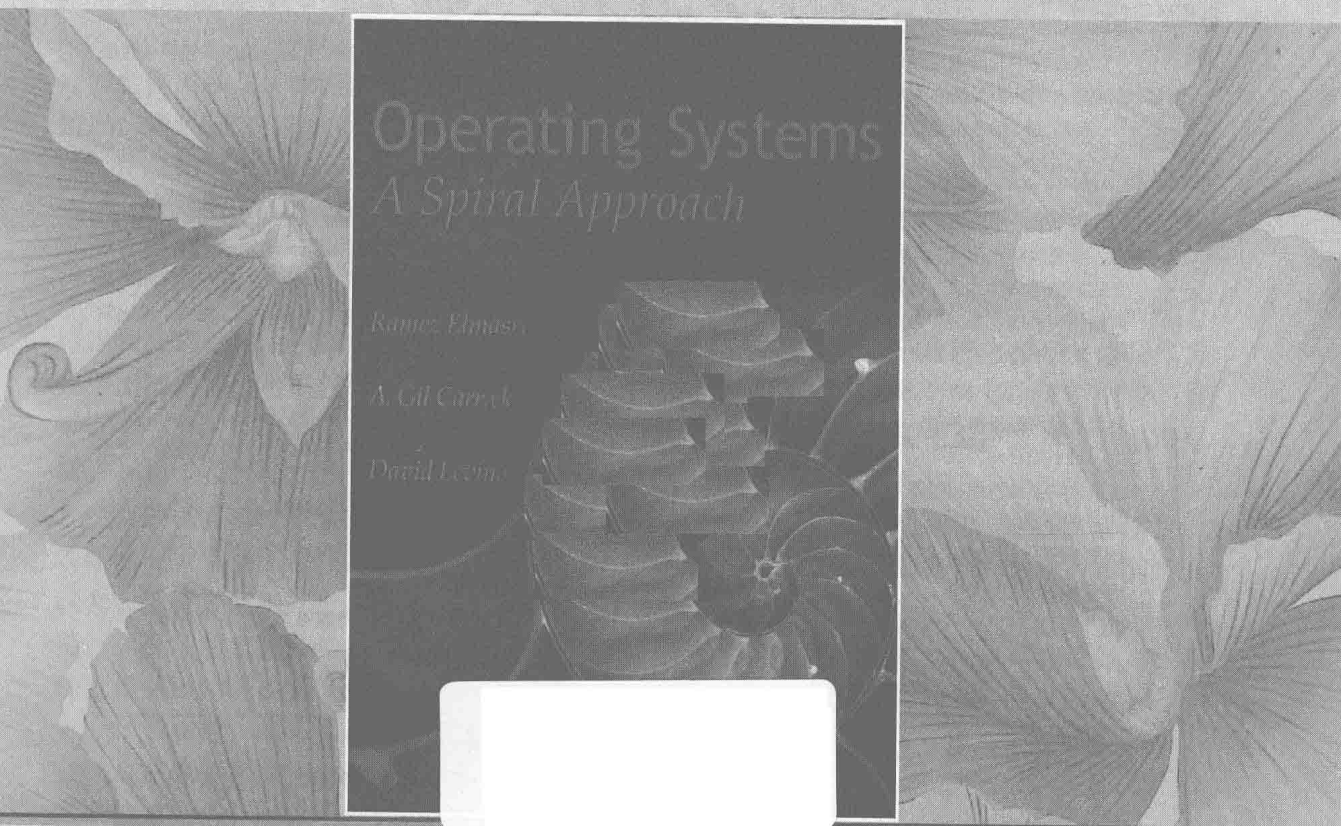
计 算 机 科 学 丛 书

操作系统实用教程

螺旋方法

拉米兹·埃尔玛斯瑞 A. 吉尔·卡里克 戴维·莱文
(Ramez Elmasri) (A. Gil Carrick) (David Levine) 著
[美] 得克萨斯大学阿灵顿分校
翟高寿 译

Operating Systems
A Spiral Approach



机械工业出版社
China Machine Press

图书在版编目(CIP)数据

操作系统实用教程：螺旋方法 / (美) 拉米兹·埃尔玛斯瑞 (Ramez Elmasri) 等著；翟高寿译. —北京：机械工业出版社，2018.1

(计算机科学丛书)

书名原文：Operating Systems: A Spiral Approach

ISBN 978-7-111-58819-1

I. 操… II. ①拉… ②翟… III. 操作系统—教材 IV. TP316

中国版本图书馆 CIP 数据核字 (2017) 第 316911 号

本书版权登记号：图字 01-2009-6672

Ramez Elmasri, A. Gil Carrick, David Levine: Operating Systems: A Spiral Approach (978-0-07-244981-5)

Copyright © 2010 by McGraw-Hill Education.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including without limitation photocopying, recording, taping, or any database, information or retrieval system, without the prior written permission of the publisher.

This authorized Chinese translation edition is jointly published by McGraw-Hill Education and China Machine Press. This edition is authorized for sale in the People's Republic of China only, excluding Hong Kong, Macao SAR and Taiwan.

Copyright © 2018 by McGraw-Hill Education and China Machine Press.

版权所有。未经出版人事先书面许可，对本出版物的任何部分不得以任何方式或途径复制或传播，包括但不限于复印、录制、录音，或通过任何数据库、信息或可检索的系统。

本授权中文简体字翻译版由麦格劳-希尔(亚洲)教育出版公司和机械工业出版社合作出版。此版本经授权仅限在中华人民共和国境内(不包括香港、澳门特别行政区及台湾地区)销售。

版权 © 2018 由麦格劳-希尔(亚洲)教育出版公司与机械工业出版社所有。

本书封面贴有 McGraw-Hill Education 公司防伪标签，无标签者不得销售。

本书是一本特色鲜明的操作系统教材，采用螺旋式方法由浅入深、循序渐进地阐述操作系统的相关概念和设计机理。全书共分六部分：第一部分阐释操作系统的概念、视图、起源、分类、构建方法及实现基础；第二部分则渐进地介绍单进程操作系统、单用户多任务操作系统、单用户多任务/多线程操作系统、多用户操作系统、分布式和集群及网络相关的操作系统；第三部分和第四部分则按照传统方式集中讨论进程管理、内存管理、文件系统、输入/输出管理；第五部分介绍计算机网络、保护和安全以及分布式操作系统；第六部分则分别就 Windows NT 操作系统、Linux 操作系统、Palm 操作系统进行实例研究。另外，附录部分还简要介绍了比较现代的硬件体系结构的相关知识。每一章结尾部分均配备有习题，可以帮助读者有针对性地加强相应知识的理解。

本书适合作为高等院校计算机及相关理工科专业的操作系统课程教材，同时也可作为业界人士设计和开发操作系统及相关系统软件的重要参考书。

出版发行：机械工业出版社(北京市西城区百万庄大街 22 号 邮政编码：100037)

责任编辑：朱秀英

责任校对：殷虹

印刷：北京诚信伟业印刷有限公司

版次：2018 年 1 月第 1 版第 1 次印刷

开本：185mm×260mm 1/16

印张：30

书号：ISBN 978-7-111-58819-1

定价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅筹划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

华章网站：www.hzbook.com

电子邮件：hzsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



华章教育

华章科技图书出版中心

由拉米兹·埃尔玛斯瑞 (Ramez Elmasri)、A. 吉尔·卡里克 (A. Gil Carrick)、戴维·莱文 (David Levine) 编写的这本书是一本以学生为中心的具有鲜明特色的操作系统教材, 采用螺旋式方法由浅入深、循序渐进地阐述操作系统的相关概念和设计机理。相比于传统教材所采用的深度导向的阐释方法, 本书的螺旋式方法更符合人们的学习规律, 也更有利于读者在学习过程中不断积聚信心, 并顺理成章地理解和掌握越来越复杂的操作系统概念及相关方法与技术, 从而保证学习效果和质量。与此同时, 作者写作经验丰富, 任教阅历和操作系统相关领域的工作经验丰富, 并乐于与学生们畅谈操作系统当前研究状况和研习操作系统最新进展, 这为教材的高质量编写创造了条件和基础。我们翻译这本教材, 目的是期望对国内大学的操作系统课程教学方法的改革和实践注入新鲜血液及提供适合的物质基础, 改善学生对操作系统设计原理的理解和掌握力度, 进而为计算机操作系统这一核心、基础系统软件的设计和开发培养高质量的人才贡献绵薄之力。

具体而言, 这本教材具有如下几方面的特色和优势:

1) 注重教材编写与学习规律的一致性, 在介绍操作系统概念和设计机理的过程中, 始终贯彻由浅入深、由易到难的编排次序, 从而使复杂概念和问题的理解和掌握建立在相关基础性概念和问题均已理解和掌握的前提下, 于是所谓的“复杂”已不再那么复杂, 学习者自我学习信心和学习兴趣逐渐增强, 学习效果和质量得到了保证。

2) 教材编写图文并茂, 大量的图示可以帮助那些更擅长形象思维的学习者更好地理解、掌握相关知识点和内容, 同时文字编写常常采用设问、讨论等会话式写作风格, 从而避免了过度学究式说教风格让学生产生厌学情绪和影响课程学习效果, 而且通过多种学习模式的支持还可保证学习质量和提高学习效果。

3) 把相关操作系统重新还原到当时具有实际时代意义的历史环境中, 有助于学生更好地理解操作系统的相关知识。因此, 对于单独分析讨论的各种操作系统, 无论在螺旋式教学章节还是在实例研究部分, 教材也会介绍当时工业发展的一些史实, 有时甚至会提到主要的公司或个人, 这对于学生全面理解操作系统概念和技术的起源及需求, 甚至对于学生创新思维的培养将具有非常重要的推动作用。

4) 教材资源丰富, 为教学的顺利开展奠定了坚实的基础。每一章结尾部分均配有习题, 可以帮助读者有针对性地加强对相应知识的理解和掌握。全书最后附有索引, 可方便读者快速查阅相关概念和关键术语。服务于本教材的配套网站提供了课件, 方便学生预习使用, 也方便教师开展课堂教学时在此基础上加以裁剪和修正使用, 同时提供给教师的习题答案还可方便教师批阅作业时参考。另外, 配套网站为许多章节还配备有实验课题, 可方便教师布置实验设计作业并使学生在实践中强理解。

本书适合作为学时安排为一个学期且为大学二年级以上的学生开设的操作系统本科课程的教材。

本书的出版得到了机械工业出版社华章公司和北京交通大学的大力支持, 在此表示最真诚的感谢! 特别要感谢曲熠老师为本书的出版所做的辛苦工作和努力! 同时非常感谢朱秀英

编辑对译稿文字的精心校对和中肯建议！特别感谢唐晓琳编辑对译稿排版组织的精美设计和对译文质量的精雕细琢！

翟高寿对全书所有内容进行了翻译、审阅和校正。以下人员提供了部分章节的译文草稿：刘晨（第15章和第19章），任艳艳（第17章和第18章），孙浩林（第16章和第20章），贾靖仪（第8章和附录），朱雪燕（第6章和第11章），翟瑞霞（第7章、第10章和第13章），王佳（第5章和第9章），韩梦梦（第12章和第14章）。在此对他们的贡献表示诚挚的感谢。

尽管译者已经反复阅读和审校译文，并竭尽全力地以“信、达、雅”为标准来再现原作者的论述本意和写作水平，但鉴于译者水平有限，难免仍有疏漏或不当之处，欢迎各位专家和广大读者批评指正。

翟高寿

2017年10月于北京交通大学

著书起因

长期以来,我们深刻领会到传统的操作系统课程教学方法并非最好的方法。本书将采用非传统教学方法来支持达成相关教学任务。当学习任何一门学问时,关于原理、规则、思想和概念的层级体系的学习次序,可能会让学习过程变得更加容易或者更为困难。最通用的做法是把课程划分为若干个主题,然后逐个具体展开和分别进行学习。就操作系统而言,传统做法就是首先概括介绍操作系统和简要解释一些术语,然后让学生就进程和进程管理、存储管理、文件系统等孤立主题分别进一步深入学习。我们称之为深度导向的教学方法或纵向型教学方法。学生在学习这些孤立主题领域大量不相关细节知识的基础上,针对具体实例和实际操作系统例子进行分析研究,最终实现不同主题的融会贯通,搞清楚如何把不同的功能模块整合到一起形成一个实用的操作系统。

我们认为,儿童学习一门语言时可遵循的较好模型应当是:首先学习一些单词、一些语法、一些句型,然后(螺旋式)重复这一过程,即不断学习更多的单词、更多的语法、更多的句型。通过螺旋式重复相同的学习过程,最终精通语言和征服语言的复杂性。我们称之为广度导向的教学方法或螺旋式教学方法。

本教材把螺旋式教学方法运用到操作系统课程教学中。前面几章给出一些基本的背景知识和概念定义。在此基础上,开始描述面向一个简单系统(早期的个人计算机)的非常简单的操作系统,然后逐渐向拥有更多功能的复杂系统演化:从有限的后台任务(譬如并发打印)到多任务,等等。对于任何一个阶段和相应的系统,我们始终坚持建立逐渐增加的需求与系统设计之间的关联关系,并阐明二者间的关联效应。当然,此间论述并不一定完全对应操作系统发展的确切的历史次序。特别是,为了能够讲清楚不同的操作系统模块之间如何相互衔接和彼此影响,我们在每个复杂层级上还选择了一个代表性的系统展开详细说明。我们确信,这种方法将有利于学生更好地理解 and 掌握操作系统每一层级的诸多功能是如何被整合到一起的。

之所以采用这种方法,在一定程度上也和所有计算科学专业学生均无一例外地要求必修操作系统课程存在一定关系。诚然,这些学生中的大部分将来从事操作系统开发工作的可能性微乎其微,然而,除极个别学生的工作可能与没有操作系统的嵌入式系统打交道外,大多数学生所从事的工作依赖的系统将运行在操作系统平台之上。对于他们而言,操作系统位于应用程序和硬件之间,若不能清晰理解操作系统基本知识,将意味着相关应用程序最好不过是低效运行,最坏情况下甚至危险运行。我们相信,相对于传统方法来讲,我们的方法将有助于引导学生更好地理解 and 掌握现代操作系统的整体结构。

内容组织

在本书的第一部分,我们给出了一些通常的背景知识。它们涵盖操作系统的基本原理,并从不同角度诠释了操作系统。同时,还概括说明了操作系统所控制的典型计算机硬件。另外一章则阐述了进程、多道程序设计、分时、资源管理等概念及不同的操作系统体系结构和

构建方法。

在本书的第二部分，我们采用螺旋式方法，按照复杂性逐渐递增的次序，依次介绍和说明了如下 5 种类型的操作系统：

- 1) 简单的单进程操作系统 (CP/M)。
- 2) 允许简单系统多任务化的较为复杂的操作系统 (Palm 操作系统)。
- 3) 针对单用户的完全多任务化的操作系统 (苹果电脑 Mac 操作系统, OS X 前身)。
- 4) 多用户操作系统 (Linux)。
- 5) 分布式操作系统 (主要是 Globus 集群)。

针对每一种操作系统，我们分别选择典型的操作系统展开讨论，使相关知识更加具体明了。典型操作系统的选择兼顾了实用性。我们首先从进程、内存、文件和输入/输出管理等方面讨论了简单的系统，然后通过逐渐引入多任务、分时、网络连接、安全和其他问题来(循序渐进地)讨论更为复杂的系统。有时候我们也会提到其他一些众所周知的操作系统作为特定类型的例子，譬如第 3 章的微软 DOS 操作系统 (MS-DOS) 和第 4 章的 Symbian 操作系统。

在本书的第三~五部分，我们转向深度导向的教学方法，针对各种操作系统主题(从进程到内存管理，再到文件系统)展开详细说明。我们还讨论了许多在操作系统领域最近才出现的热点问题，如线程化、面向对象、安全以及并行和分布式系统的相关方法。在相应章节中，我们将重温第二部分曾讨论过的实例系统，并进一步详细解释(尤其是现代操作系统的)相关机制。

在第六部分，我们将以所谓实例研究的方式，就一些操作系统进一步展开深入探讨。鉴于我们已经在前面章节介绍了许多细节内容，所以在此我们将立足于从更深层次审视相关系统，从而探析某些功能的内部实现机制。其中的两个实例研究将围绕第二部分涵盖的操作系统进行分析讨论。

附录部分罗列了基本的计算机硬件体系结构，以方便那些不要求把此类课程作为操作系统课程先修课程的教育机构在选用本教材时使用。它们也可为那些需要复习特定主题的学习者提供参考。

写作风格

- 我们倡导会话式写作风格，以避免过度的学术式说教风格让学生产生厌学情绪和影响课程学习效果。
- 我们避免使用过多的形式化描述，较为规范描述仅用于必需的特定场合。这样做的理由在于，大多数学生将来并不会从事开发操作系统的工作，而往往是基于操作系统来支持应用程序的开发、运营或维护。
- 一般情况下，我们使用规范的、公认的术语。但是，当不存在公认的标准术语或者着眼于说明特定历史时期采用的其他术语时，我们也会讨论其他说辞或用语。
- 通常，我们讨论算法解决方案而不列出实际代码，因为不同学校的学生往往接触和掌握的是不同的编程语言。
- 我们认为，如果把相关操作系统重新还原到当时具有实际时代意义的历史背景环境中，将有助于学生更好地理解操作系统的相关知识。因此，对于单独分析讨论的各操作系统，无论在螺旋式教学章节还是在实例研究部分，我们也会介绍当时工业发

展的一些史实，有时甚至会提到主要的公司或个人。

- 考虑到学生天天都在使用按照惯例并不被视作计算机的各种装置，而且对它们的操作非常熟悉，所以本教材内容涵盖这些便携式装置上的现代操作系统。
- 我们在教材中使用了大量的图示，以帮助那些不喜欢啃读词句而更擅长形象思维的学习者更好地理解和掌握相关知识点和内容。
- 各章结束部分分别配备一套习题，方便学习者用于评估自己关于对应章节内容的理解和掌握情况。
- 许多章节还配备有实验课题，可方便教师布置实验设计作业并使学生在实践中强化理解。

如何使用本教材：对教师的建议

这本教科书可用于学时安排为一个学期且为大学三年级或者四年级的学生开设的操作系统本科课程。本书的第一部分着眼于为后续各章节奠定必需的基本背景知识。第1章主要采用设问和讨论的叙述方式，并提供基于历史视角的若干观点。任课教师可以略读这一章并决定包含哪些教学内容。附录部分则是对比较现代的硬件体系结构的简要介绍。如果硬件课程没有作为本课程的先修课程，那么建议把附录内容包含在教学过程中。第2章定义了操作系统用到的一些简单术语，并提供了关于操作系统设计这一更大主题的更多观点。同样，任课教师可以对这一章的内容进行复习，并选择在教学中包含或者排除某部分内容。

第二部分采用了螺旋式方法。我们认为，这是本书最具特色的部分。在这一部分，学生可以渐进式地领略到一系列设计目标越来越复杂的操作系统。其中只有两章内容在操作系统教科书中属于不同寻常的主题，即关于单用户多任务操作系统的第4章和关于分布式系统的第7章。这两章可由教师自行决定是否跳过，不过必须指出，越来越多的学生将会以用户身份或者程序员角色工作在那样的系统环境中。

第三~六部分则着眼于对操作系统的深度剖析。在这些部分，除了第12章和第13章彼此之间密切相关以外，其余各章均相对独立。另外，对于从第14章开始的各章来说，如果对应标题是要求学生选修的另一课程的主题，那么相应章节可以跳过。

请注意有关的参考文献：各章所引用的文献往往被广泛认为是比较重要的资料或者是很好的总结，而且有可能涵盖本教材所没有提到的内容。如果教师或学生想要查阅材料以便能够更好地理解特定主题，那么建议研读相应的文献。

如何使用本教材：对学生的建议

对于学生来说，使用这本教科书最重要的事情就是要搞清楚如何才能学得最好。让大脑牢记信息的途径可以有好多种。本教科书自身就直接提供了两种这样的途径。显而易见，本书图文并茂，既适用于那些擅长咬文嚼字式读书的人，又适用于那些倾向于视觉型学习风格的人。当参加课堂听讲时，你将聆听到教师对教材内容的讲解，这适用于那些擅长通过听讲来获取知识的人。与此同时，任课教师也可以使用配合本教科书提供的课件等直观教具。同样，这对于那些擅长通过读书和看图来学习知识的人来说也是非常有益的。有些学生擅长机械地学习，故而对于那些学生来说，概括教材内容或者做好学习笔记往往可以获得不错的学习效果。

与其他教材相同，本书在各章的结尾部分同样也提供了针对相关内容的习题。这些习题是这样设计的：对于那些已经掌握了有关教学内容的学生来说应当能够正确回答相应的问题。

在新的知识进入大脑的过程中，往往需要花费一定的时间才能够和原有的其他信息有机地关联起来。然而，如果一天内大脑接受了很多信息，那么那些不太重要的信息大脑常常不会保留，只有在相同或相似信息间隔不长时间就再次呈献给大脑时，大脑才会记住相应信息。采用的不同机制越多、有关信息的重复次数越多，对应教学内容的保持才会越持久和

越牢固。因此,最好的学习方法就是综合运用各种方法,并重点采用适合你和你所擅长的方法。我们发现,对于大多数学生来说,采用如下的学习方式或次序往往会取得很好的学习效果:

- 提前打印涵盖下一节教学内容的课件(每页若干张幻灯片)。
- 阅读教科书中的指定内容,并特别注意所打印课件中给定的问题。
- 进入课堂聆听教师讲解,认真做笔记,特别是教师提到的不在教科书上的内容。(相关知识点往往是教师喜欢的问题,而且常常会出现在考题中。)
- 对于任何不清楚的地方要及时提问和请教。
- 在准备考试和复习教学内容的时候,重新过一遍课件。如果还有某些知识点不太清楚,要找到教科书上对应章节彻底搞清楚和确保真正掌握。如果还有疑问,那么请联系教师和助教加以解决。
- 在学生认为方便的任何时候对相关习题进行研究。

为教师提供的资源^①

本教科书通过一个网站为教师和学生分别提供了彼此分离和独立的支持:

- 只要发现有需要,就会不时地为本教科书提供补充材料。
- 为教师提供了一组建议性的实验项目课题,其中大多数的实验项目作者将会在教学过程中加以使用。这些项目足够丰富且与特定操作系统平台是无关的,故而能够方便地加以调整和适用于任何教学场景。它们不依赖于任何特定的软件包,因此不要要求教师、学生或助教开展相关实验前加以掌握。
- 正如前面所提到的,为学生提供了可用的课件。对于教师而言,则鼓励对这些课件进行修改以满足特定的需求,同时要求给出相应课件的出处并给以致谢。
- 为教师提供了习题答案,以便于他们不会因为不太清楚某些晦涩难懂的但作者认为重要的知识点而不好意思或陷入尴尬境地。
- 在有关网站上维护一个当前最新的勘误表。
- 许多章节提供了所参考的网上资源,但考虑到相关网站极易发生改变,所以本教科书的支持网站将会保持一份最新版的网上资源信息。

致谢

事实上,本教科书已经构思相当长的时间了,比我们可以记起的时间还要长。麦格劳-希尔(McGraw-Hill)出版社对我们给予了格外的耐心。特别是,我们愿意把最诚挚的谢意献给麦格劳-希尔出版社的如下同仁:梅林达·毕莱基(Melinda Bilecki)、凯·布莱迈耶(Kay Brimeyer)、布伦达·罗尔维斯(Brenda Rolwes)、卡拉·库卓恩维茨(Kara Kudronowicz)、费伊·仙凌(Faye Schilling)以及拉古·斯里尼瓦桑(Raghu Srinivasan)。同时,我们也要感谢曾在我们启动本书写作时担任编辑的艾伦·阿普特(Alan Apt)和艾米丽·陆帕什(Emily Lupash)。最后,我们还要感谢埃里卡·乔丹(Erika Jordan)和劳拉·潘池考夫斯基(Laura Patchkofsky)对松树构图的创意。

① 关于本书教辅资源,用书教师可向麦格劳-希尔教育出版公司北京代表处申请,电话:8008101936/010-62790299-108,电子邮件:instructorchina@mcgraw-hill.com。——编辑注

第 18 章关于“Windows Vista”的内容由微软公司的戴夫·普罗伯特 (Dave Probert) 进行了审阅。针对我们只能进行推测的一些事项以及由此引起我们关注的若干问题, 他为我们提供了非常有价值的反馈建议和意见。他的参与是由阿尔卡季·瑞蒂克 (Arkady Retik) 与微软公司协同安排的。另有两章是由我们在得克萨斯大学阿灵顿分校的同事审阅的。具体而言, 刘永和 (Yonghe Liu) 审阅了第 15 章, 而马修·赖特 (Matthew Wright) 则审阅了第 16 章。另一位教师巴赫拉姆·哈利利 (Bahram Khalili) 在他的操作系统课上使用了本教科书的草稿。当然, 若有任何遗留问题, 责任在于我们, 而与他们没有关系。

还有, 我们已经利用这些素材的草稿好多年了。在此, 我们希望表达对我们的所有学生以及他们所反馈的意见和建议的感谢。特别是, 我们要感谢如下学生: 扎希尔·纳莱恩 (Zaher Naarane)、菲尔·伦纳 (Phil Renner)、威廉·皮科克 (William Peacock)、韦斯·帕里什 (Wes Parish)、凯尔 D. 威特 (Kyle D. Witt)、戴维 M. 康奈利 (David M. Connelly) 以及斯科特·珀迪 (Scott Purdy)。

关于遗留错误的声明

多名作者协作完成一项计划的困难之一在于, 某位作者本来是好心, 然而却可能修改了并非他自己所写的一些文字, 尽管他自己认为是在澄清一些小的问题, 但是却以某种不易察觉而非常重要的方式修改了相应的含义。因此, 你可能会发觉教材中还存留某些小问题。当然, 这些错误并非是原先作者的错误。毋庸置疑, 原先的作者准确无误地完成了最初的文稿, 相关错误可能是另一满怀善意但却对相关材料不很熟悉的作者引入的。

无论如何, 总可能会存在这样一些错误, 所以我们必须要认真地加以处理。因此, 如果你发现了错误, 我们非常高兴和希望知道它们是什么错误。我们将会发布勘误表, 在下一版中予以修正, 确定责任人员, 并适当处理出错的作者。

作者介绍

Operating Systems: A Spiral Approach

我们使用其他教材讲授操作系统课程已经好多年了。由于希望采用另一种不同的教学方法，我们编写了这本教材。我们全都是得克萨斯大学阿灵顿分校（University of Texas at Arlington, UTA）计算机科学与工程系的教师。

拉米兹·埃尔马斯瑞（Ramez Elmasri） 得克萨斯大学阿灵顿分校教授。1972年在埃及亚历山大大学电气工程专业获学士学位，1980年在斯坦福大学计算机专业获硕士学位和博士学位。他当前的研究兴趣包括传感器网络、射频识别、生物信息学数据介质、查询个性化以及系统集成。他是教材《Fundamentals of Database Systems》（数据库系统基础）的第一作者，该教材已发行至第5版。他以往的研究涵盖数据库、概念建模和分布式系统的各个方面。

A. 吉尔·卡里克（A. Gil Carrick） 以前是得克萨斯大学阿灵顿分校讲师，现已从教师岗位退休。1970年在休斯敦大学电子技术专业获学士学位，2000年在得克萨斯大学阿灵顿分校计算机专业获硕士学位。他是计算机科学荣誉学会的成员。他的职业跨越整个信息技术产业，包括终端用户组织、硬件制造商、软件出版商、第三方维护机构、大学以及研发公司。他为专业期刊撰稿，并编辑信息技术书籍，相关选题主要集中在网络领域。在他的职业生涯中，这本教材中所讨论的所有操作系统他都使用过，他甚至还使用过许多其他的操作系统。

戴维·莱文（David Levine） 讲授操作系统、软件工程、网络和计算机体系结构课程。他的研究兴趣包括移动计算、移动对象和分布式计算，整理的相关研究成果发表在近几年的出版物和若干国际会议上。他喜欢讨论操作系统，与学生畅谈操作系统的当前研究，并研习操作系统的最新进展。

出版者的话	
译者序	
前言	
教材使用说明	
作者介绍	

第一部分 操作系统概述

第1章 入门	2
1.1 引言	2
1.2 什么是操作系统	3
1.3 操作系统的用户视图和系统视图	4
1.3.1 用户视图及用户分类	4
1.3.2 系统视图	5
1.3.3 一个例子：移动鼠标 (和鼠标指针)	6
1.3.4 另一个比较大的例子： 文件	7
1.4 操作系统的一些术语、基本概念和 图解	7
1.4.1 基本术语	7
1.4.2 这些图片说明了什么	8
1.4.3 走近真实：个人计算机操作 系统	9
1.4.4 为什么设立抽象层	10
1.5 操作系统发展导论	11
1.5.1 操作系统的起源	11
1.5.2 操作系统应当做什么	12
1.6 小结	13
习题	13

第2章 操作系统概念、模块和 体系结构

2.1 操作系统做什么工作	14
---------------	----

2.2 操作系统管理的资源及主要的 操作系统模块	16
2.2.1 操作系统管理的资源类型	16
2.2.2 操作系统的主要模块	18
2.3 进程概念和操作系统进程信息	19
2.3.1 进程定义和进程状态	19
2.3.2 操作系统维护的进程信息	21
2.3.3 进程分类和执行模式	21
2.4 面向功能的操作系统分类	22
2.4.1 单用户单任务操作系统	22
2.4.2 多任务操作系统	22
2.4.3 分时操作系统和服务	23
2.4.4 网络和分布式操作系统	24
2.4.5 实时操作系统	25
2.5 操作系统构建方法	25
2.5.1 整体式单内核操作系统方法	25
2.5.2 分层式操作系统方法	25
2.5.3 微内核操作系统方法	26
2.6 操作系统实现中的一些问题和 技术	27
2.6.1 基于中断向量的中断处理	27
2.6.2 系统调用	28
2.6.3 队列和表	28
2.6.4 面向对象的方法	29
2.6.5 虚拟机	29
2.7 操作系统功能及向后兼容的 最小化方法和最大化方法	31
2.7.1 向后兼容	31
2.7.2 用户最优化与硬件最优化	32
2.8 小结	32
参考文献	32
网上资源	33
习题	33

第二部分 渐进式构建操作系统： 面向广度的螺旋式方法

第3章 简单的单进程操作系统..... 37

3.1 监控程序和 CP/M.....	37
3.1.1 监控程序：简单操作系统的前身.....	37
3.1.2 为什么创建 CP/M？什么是软件危机.....	38
3.1.3 CP/M 的构成.....	39
3.2 简单的个人计算机系统的特征.....	39
3.3 输入 / 输出管理.....	40
3.3.1 键盘输入——可移植性与灵活性.....	41
3.3.2 视频监视器输出——可移植性及功能与性能.....	41
3.4 磁盘管理和文件系统.....	42
3.4.1 磁盘系统.....	42
3.4.2 文件系统.....	43
3.5 进程和内存管理.....	46
3.5.1 应用程序的创建与执行.....	46
3.5.2 基于 CCP 的命令处理.....	47
3.5.3 内存管理.....	48
3.5.4 覆盖.....	49
3.5.5 进程及基本的多任务.....	49
3.6 小结.....	50
参考文献.....	50
网上资源.....	51
习题.....	51

第4章 单用户多任务操作系统..... 52

4.1 简单的多任务系统.....	53
4.2 Palm 操作系统运行环境及系统布局.....	54
4.2.1 基本内存为易失性随机访问存储器.....	55
4.2.2 没有辅助存储器.....	55
4.2.3 小屏幕尺寸.....	55
4.2.4 没有键盘.....	56
4.3 进程调度.....	56

4.3.1 处理涂鸦式输入——实时操作系统任务.....	56
4.3.2 应用程序进程——任何时候只能有一道进程持有焦点.....	57
4.3.3 典型的用户应用程序.....	57
4.3.4 真正的调度程序开始成形.....	58
4.4 内存管理.....	58
4.4.1 内存基础知识.....	58
4.4.2 内存分配.....	59
4.4.3 不可移动的内存块.....	61
4.4.4 空闲空间监测.....	61
4.5 文件支持.....	62
4.5.1 数据库和记录.....	62
4.5.2 资源对象.....	62
4.5.3 辅助存储器.....	63
4.6 基本输入 / 输出.....	63
4.7 显示管理.....	64
4.7.1 相应硬件.....	64
4.7.2 高级图形化用户界面元素.....	64
4.7.3 特殊的窗体类型.....	64
4.7.4 低级图形化用户界面控件.....	65
4.8 事件驱动的程序.....	66
4.9 小结.....	67
参考文献.....	67
网上资源.....	67
习题.....	68

第5章 单用户多任务 / 多线程操作系统..... 69

5.1 引言.....	69
5.2 Mac 计算机的起源.....	69
5.3 Mac 操作系统——第 1 版系统.....	70
5.3.1 图形化用户界面.....	70
5.3.2 单任务.....	71
5.3.3 辅助存储器.....	72
5.3.4 内存管理.....	72
5.3.5 只读存储器.....	74
5.3.6 增量版本.....	74
5.4 第 2 版系统.....	74
5.4.1 图形化用户界面.....	75

5.4.2 多任务	75
5.5 第3版系统	75
5.5.1 多级文件系统	75
5.5.2 网络	76
5.6 第4版系统	76
5.6.1 多重查找器	76
5.6.2 多重查找器与图形化用户界面	77
5.6.3 内存管理与多重查找器	77
5.7 第5版系统	78
5.8 第6版系统	78
5.9 第7版系统	79
5.9.1 图形化用户界面	79
5.9.2 虚拟内存	79
5.9.3 新型处理器	80
5.9.4 输入/输出增强	81
5.10 第8版系统	82
5.10.1 多级文件系统升级版	82
5.10.2 其他的硬件变化	83
5.10.3 统一字符编码标准支持	83
5.11 第9版系统	84
5.11.1 多用户	84
5.11.2 网络	85
5.11.3 应用程序接口	85
5.11.4 视频	86
5.12 X版Mac操作系统	86
5.12.1 新功能	87
5.12.2 又一款新处理器	87
5.13 小结	87
参考文献	87
网上资源	88
习题	88

第6章 多用户操作系统 90

6.1 引言	90
6.1.1 多用户操作系统的历史	90
6.1.2 Linux操作系统的基本结构	93
6.1.3 动态可加载模块	94
6.1.4 中断处理	95
6.1.5 文件系统目录树	96
6.2 多用户操作系统环境	96

6.2.1 文件访问权限	97
6.2.2 文件控制块	98
6.3 进程和线程	98
6.3.1 Linux任务	98
6.3.2 抢占式多任务	99
6.3.3 对称多处理	99
6.4 小结	101
参考文献	101
网上资源	101
习题	101

第7章 并行分布式计算、集群和网格 102

7.1 引言	102
7.2 关键概念	102
7.3 并行处理和分布式处理	103
7.4 分布式系统体系结构	105
7.4.1 执行环境概述	105
7.4.2 对称多处理系统	106
7.4.3 集群	107
7.4.4 计算网格	108
7.4.5 志愿计算	109
7.5 操作系统相关概念在对称多处理、集群和网格中的解读	111
7.5.1 进程同步和通信	111
7.5.2 一个例子	111
7.5.3 例子复杂化的一面	112
7.5.4 对称多处理的解决方案	112
7.5.5 集群的解决方案	112
7.5.6 网格的解决方案	112
7.5.7 文件共享技术	113
7.5.8 远程服务的运用	114
7.5.9 故障处理	114
7.6 举例说明	115
7.6.1 在集群和网格上的科学计算	115
7.6.2 人类基因组脱氧核糖核酸 组装	115
7.6.3 IBM计算生物学中心和 集群计算	116
7.6.4 志愿计算集群	116

7.6.5 一个典型的计算机集群	117
7.6.6 Globus 集群的使用	117
7.6.7 门户和万维网界面	118
7.7 小结	119
参考文献	119
网上资源	119
习题	119

第三部分 处理器管理及内存管理

第8章 进程管理：概念、线程和

调度

8.1 引言	122
8.2 进程描述符——进程控制块	123
8.3 进程状态和转换	123
8.4 进程调度	126
8.4.1 先来先服务调度	126
8.4.2 优先级调度	126
8.4.3 保证型调度	127
8.4.4 最短运行时间优先调度	127
8.4.5 高响应比优先调度	128
8.4.6 抢占式调度	128
8.4.7 多级队列调度	129
8.4.8 最佳算法的选择	130
8.4.9 长程调度器	132
8.4.10 处理器亲和性	133
8.5 进程创建	133
8.6 线程	135
8.6.1 什么是线程	135
8.6.2 用户级线程与内核级线程	136
8.6.3 线程支持模型	137
8.6.4 同时多线程	139
8.6.5 进程与线程	139
8.7 实例研究	139
8.7.1 POSIX 线程	140
8.7.2 Windows NT	140
8.7.3 Solaris	142
8.7.4 Linux	142
8.7.5 Java	143
8.8 小结	144

参考文献	144
网上资源	144
习题	144

第9章 进程管理进阶：进程间通信、同步和死锁

9.1 为什么会有协作式进程	146
9.2 进程间通信	148
9.2.1 通信机制的特性	149
9.2.2 进程间通信系统的例子	152
9.2.3 共享内存系统的例子	154
9.3 同步	154
9.3.1 相关问题	154
9.3.2 原子操作	155
9.3.3 锁与临界区	155
9.3.4 硬件锁指令	156
9.3.5 信号量与等待	157
9.3.6 计数型信号量	157
9.3.7 同步与流水线体系结构	158
9.3.8 对称多处理系统中的同步	158
9.3.9 优先级倒置	158
9.3.10 经典问题	159
9.3.11 管程	160
9.4 死锁	161
9.4.1 什么是死锁	161
9.4.2 可以对死锁采取哪些措施	163
9.4.3 死锁预防	163
9.4.4 死锁避免	165
9.4.5 死锁检测	167
9.4.6 抢占和其他实用的解决方案	167
9.5 小结	168
参考文献	168
网上资源	169
习题	169

第10章 基本的内存管理

10.1 为什么要管理主内存	171
10.2 开发周期步骤与绑定模型	171
10.3 单一进程	172
10.3.1 编码时绑定	172

10.3.2	链接时绑定	173
10.3.3	单一进程	174
10.3.4	动态重定位	174
10.3.5	物理内存空间与逻辑内存空间	175
10.3.6	程序比内存大	175
10.3.7	覆盖	175
10.3.8	对换	176
10.4	固定进程数的多进程	177
10.4.1	内部碎片	178
10.4.2	分时共享	178
10.5	可变进程数的多进程	178
10.5.1	动态加载	181
10.5.2	动态链接库	181
10.6	小结	182
	参考文献	183
	习题	183
第 11 章	高级的内存管理	184
11.1	为什么需要硬件的辅助支持	184
11.2	分页	184
11.2.1	要求两次访问内存	186
11.2.2	有效的内存访问时间	186
11.2.3	内存访问控制	188
11.2.4	大页表	188
11.2.5	反置页表	190
11.2.6	支持页面大小多样化的页表	190
11.2.7	历史的注脚	191
11.3	分段	191
11.4	段页式	193
11.5	请求分页	194
11.5.1	请求分页方式下的有效内存访问时间	195
11.5.2	工作集与页面置换	196
11.5.3	脏页	198
11.5.4	其他的页面淘汰算法	199
11.5.5	每个进程应当拥有多少页面	200
11.5.6	页面限值自动平衡	200
11.5.7	抖动	201
11.5.8	页面锁定	201

11.5.9	页面清理机制	202
11.5.10	程序设计与缺页率	202
11.6	特殊的内存管理主题	203
11.6.1	进程间内存共享	203
11.6.2	内存映射文件	204
11.6.3	Windows XP 预提取文件	205
11.6.4	Symbian 内存管理	206
11.7	小结	207
	参考文献	207
	网上资源	207
	习题	207

第四部分 面向深度的操作系统概念的展示：文件系统和输入 / 输出

第 12 章	文件系统基础	210
12.1	引言	210
12.2	目录	211
12.2.1	逻辑结构	211
12.2.2	物理结构	214
12.2.3	目录操作	214
12.2.4	文件系统元数据	216
12.3	存取方法	216
12.3.1	顺序存取	216
12.3.2	随机存取	217
12.3.3	更高级别的存取方法	217
12.3.4	原始存取	219
12.4	空闲空间管理	219
12.4.1	链表式空闲空间监测	219
12.4.2	改进的链表法	220
12.4.3	位图式空闲空间监测	221
12.5	文件分配	223
12.5.1	连续分配	223
12.5.2	链接分配	225
12.5.3	索引分配	227
12.6	小结	229
	参考文献	229
	网上资源	230
	习题	230

第 13 章 文件系统实例及更多功能 ... 232

13.1 引言	232
13.2 实例研究	232
13.2.1 FAT 文件系统	232
13.2.2 NTFS 文件系统	234
13.2.3 UNIX 和 Linux 的文件系统	235
13.3 挂载	236
13.3.1 本地文件系统挂载	236
13.3.2 远程文件系统挂载	237
13.4 多文件系统和重定向	238
13.4.1 虚拟文件系统	238
13.4.2 网络文件系统	239
13.5 内存映射文件	240
13.6 文件系统实用例程	240
13.7 日志式文件系统	242
13.8 小结	243
参考文献	243
网上资源	243
习题	243

第 14 章 磁盘调度和输入 / 输出管理 ... 244

14.1 引言	244
14.2 设备特性	244
14.2.1 随机存取与顺序存取	244
14.2.2 设备分类	245
14.3 输入 / 输出技术	246
14.3.1 缓冲技术	246
14.3.2 高速缓存技术	247
14.3.3 针对短小记录的分块技术	247
14.4 磁盘物理组织	248
14.4.1 扇区、磁道、柱面及磁头	248
14.4.2 扇区分组区和扇区编址	249
14.4.3 低级格式化	250
14.4.4 速度：寻道、传输及缓冲	250
14.5 磁盘逻辑组织	251
14.5.1 分区	251
14.5.2 引导块	252
14.5.3 错误检测与校正	253
14.6 廉价磁盘冗余阵列	254
14.6.1 廉价磁盘冗余阵列构型	255

14.6.2 廉价磁盘冗余阵列故障	257
14.7 磁盘操作调度	259
14.7.1 先来先服务调度算法	259
14.7.2 搭便车调度算法	260
14.7.3 最短寻道时间优先调度算法	260
14.7.4 向前看调度算法	261
14.7.5 循环向前看调度算法	263
14.7.6 先来先服务 - 向前看调度 算法	264
14.7.7 N 轮向前看调度算法	264
14.7.8 Linux 调度程序	265
14.7.9 向控制器发送命令	265
14.7.10 哪种算法最好	266
14.8 内存直接存取型控制器和 磁盘硬件特征	266
14.8.1 内存直接存取型控制器	266
14.8.2 磁盘驱动器的其他特征	267
14.8.3 扇区保留和扇区迁移	267
14.8.4 自我监控报告技术	268
14.8.5 展望未来	268
14.9 小结	269
参考文献	269
网上资源	269
习题	270

第五部分 网络、分布式系统及安全

第 15 章 计算机网络 ... 272

15.1 为什么要把计算机通过网络 连接起来	272
15.2 基础知识	274
15.2.1 相关模型	274
15.2.2 局域网与广域网	275
15.2.3 拓扑结构	276
15.3 应用层协议	278
15.3.1 应用层	278
15.3.2 超文本传输协议	279
15.3.3 文件传输协议	280
15.3.4 简单邮件传输协议、 邮局协议及互联网邮件	

访问协议	280	16.1.3 蠕虫	300
15.4 传输控制协议和网际协议	281	16.1.4 间谍软件	300
15.4.1 传输层	281	16.1.5 拒绝服务攻击	301
15.4.2 网际协议寻址和路由	281	16.1.6 缓冲区溢出	302
15.4.3 名称解析	283	16.1.7 脚本和小应用程序	303
15.4.4 第6版网际协议	283	16.2 操作系统保护	304
15.4.5 公共实用例程	284	16.2.1 保护	304
15.4.6 其他协议	284	16.2.2 认证	304
15.4.7 防火墙	285	16.2.3 授权	305
15.5 数据链路层	285	16.3 策略、机制和技术	308
15.5.1 以太网	286	16.3.1 安全与保护策略	308
15.5.2 桥接与交换	286	16.3.2 系统崩溃保护：备份	308
15.5.3 令牌环	287	16.3.3 并发性保护	310
15.5.4 其他的数据链路方法	288	16.3.4 文件保护	310
15.5.5 网际协议地址到介质访问 控制地址的映射	288	16.4 通信安全	310
15.5.6 面向硬件的功能迁移	289	16.4.1 加密	311
15.6 广域网	289	16.4.2 消息摘要	312
15.6.1 帧中继	290	16.4.3 消息签名与证书	313
15.6.2 其他广域网技术	290	16.4.4 安全协议	314
15.7 物理层	291	16.4.5 网络保护	315
15.7.1 铜线规格	291	16.5 安全管理	316
15.7.2 光纤规格	292	16.6 小结	317
15.7.3 无线网络	292	参考文献	317
15.7.4 关于网络故障排查的说明	293	网上资源	318
15.8 网络管理	293	习题	318
15.8.1 简单管理工具	293	第17章 分布式操作系统	320
15.8.2 简单网络管理协议和网络 设备管理	293	17.1 引言	320
15.8.3 数据包捕获	294	17.2 分布式应用模型	322
15.8.4 远程监控	294	17.2.1 客户端-服务器模型	322
15.9 小结	295	17.2.2 三层模型	322
参考文献	295	17.2.3 N层应用程序	323
网上资源	295	17.2.4 水平分布	324
习题	296	17.3 抽象概念：进程、线程和机器	325
第16章 保护和安全的	298	17.3.1 线程	325
16.1 问题和威胁	298	17.3.2 虚拟机	326
16.1.1 计算机病毒	299	17.4 命名	327
16.1.2 特洛伊木马	300	17.4.1 发现服务和 Jini	328
		17.4.2 发现服务、X.500 以及 轻量级目录访问协议	328

17.4.3 对移动式网际协议的实体的定位	328
17.5 其他分布式模型	329
17.5.1 远程过程调用	329
17.5.2 分布式对象	331
17.5.3 分布式文档	331
17.5.4 分布式文件系统	331
17.6 同步	332
17.6.1 时钟	332
17.6.2 互斥	333
17.6.3 选举	334
17.6.4 可靠的多播通信	336
17.6.5 分布式事务	337
17.7 容错	338
17.7.1 概述	338
17.7.2 进程韧性	338
17.7.3 可靠的客户端-服务器通信	339
17.7.4 分布式提交	339
17.8 小结	340
参考文献	340
网上资源	341
习题	341

第六部分 实例研究

第 18 章 从 Windows NT 到

Windows Vista 344

18.1 Windows NT 系列操作系统发展历程	345
18.2 用户操作系统环境	349
18.3 进程调度	351
18.4 内存管理	352
18.4.1 地址空间	353
18.4.2 页面映射	353
18.4.3 分页共享及写时复制	354
18.4.4 页面置换	354
18.4.5 预取配置	355
18.5 文件支持	355
18.5.1 NTFS	356
18.5.2 NTFS 高级功能特征	360

18.6 基本输入 / 输出	362
18.6.1 分区	362
18.6.2 输入 / 输出系统分层	363
18.6.3 即插即用	363
18.6.4 设备驱动程序	364
18.6.5 磁盘类、端口及微型端口的驱动程序	365
18.7 图形化用户接口编程	365
18.8 网络	366
18.9 对称多处理	367
18.10 XP 操作系统启动速度提升措施	367
18.11 小结	368
参考文献	368
网上资源	368
习题	369

第 19 章 Linux 操作系统实例研究 370

19.1 引言	370
19.1.1 Linux 发展简史	370
19.1.2 内核体系结构	371
19.2 进程调度	372
19.2.1 实时进程	373
19.2.2 普通进程	373
19.2.3 nice 命令及相关系统调用	374
19.2.4 对称多处理负载均衡	374
19.3 内存管理	375
19.4 文件支持	376
19.4.1 标准文件系统	376
19.4.2 虚拟文件系统	377
19.4.3 进程文件系统	378
19.5 基本输入 / 输出	378
19.5.1 设备表	378
19.5.2 设备类型	379
19.6 图形化用户接口编程	382
19.7 网络	384
19.7.1 网络分层	384
19.7.2 监听连接的超级服务器	384
19.7.3 Samba	385
19.8 安全	385

19.8.1 Linux 安全模块	385	20.6.5 通信电路	394
19.8.2 网络安全	386	20.7 图形化用户接口编程	394
19.9 对称多处理	387	20.8 网络编程	394
19.10 其他 Linux 操作系统变种	387	20.8.1 个人数据同步	394
19.10.1 实时 Linux 操作系统	387	20.8.2 其他数据同步	395
19.10.2 嵌入式 Linux 操作系统	388	20.8.3 互联网应用程序	395
19.11 小结	389	20.8.4 电话应用程序	395
参考文献	389	20.9 编程环境	396
网上资源	389	20.10 类似系统和当前发展状况	397
习题	389	20.10.1 新的功能模型	397
第 20 章 Palm 操作系统实例研究	391	20.10.2 高级通信模型	398
20.1 概述	391	20.10.3 线程调度	398
20.2 多进程操作系统环境	391	20.10.4 用户界面参考设计	399
20.3 Palm 进程调度	392	20.10.5 位置感知类应用程序	400
20.3.1 实时任务	392	20.10.6 后来的 Palm 操作系统版本	400
20.3.2 其他任务	392	20.11 小结	401
20.4 Palm 内存管理	392	参考文献	401
20.5 文件支持	393	网上资源	401
20.6 输入 / 输出子系统	393	习题	401
20.6.1 音频输入 / 输出	393	附录 计算机系统总览和体系	
20.6.2 流输入 / 输出	393	结构概念	403
20.6.3 内存型磁盘驱动程序	394	索引	425
20.6.4 照相机	394		

操作系统概述

本书第一部分共包含两章：

第1章主要给出了关于“操作系统（OS）是什么？”的基本解释，阐明了操作系统为普通用户和编程人员提供的各种服务。也正是由于有了相应服务，才使得计算机的使用不再要求用户必须掌握那些低级繁琐的机器“秘笈”，而可以专注于要解决的问题。这些问题可以是任何事情，不仅包括我们通常想到的计算活动，而且还包括诸如玩游戏、动态生成艺术作品和监控汽车发动机性能等。

第2章则就操作系统概念、相关模块及体系结构提供了一个初步的高层次的概貌，同时还就学生需要了解的一般术语进行了介绍，从而为他们学习第二部分所呈现的各种越来越复杂的操作系统打下基础。

入门

操作系统（Operating System, OS）是每台计算机的核心所在。操作系统为普通用户和编程人员提供服务，从而使计算机的使用无须处理那些低级的、难以掌控的硬件指令。它为计算机操作极为广泛的各类设备——从输入/输出设备（例如打印机和数码相机）到支持计算机之间进行通信的有线或无线网络部件——提供相对统一的访问接口。它支持用户创建、管理和组织各种类型的文件。此外，大多数现代操作系统还提供了图形化用户接口（Graphical User Interface, GUI），从而使用户可以更加方便简单地操作计算机。

作为开篇第一章，我们将在第 1.1 节中简要说明操作系统的重要性和普遍性（关于后者，是指其不仅广泛使用在计算机中，而且还大量存在于我们日常生活使用的各类电子设备中）。在第 1.2 节，则从相对技术的角度出发，阐明了即便简单设备也包含操作系统的理由。接下来在第 1.3 节，我们从观察操作系统的两个角度（即用户视图和系统视图）出发，围绕“操作系统做什么？”的问题，讨论了各种不同的观点。同时，我们也讨论了各种类型的用户对操作系统的具体需求。在此基础上，还通过一些简单的例子，具体说明操作系统是如何一步步执行对应功能，从而接受和完成看起来非常简单的用户请求的。在第 1.4 节，我们提出了一些基本的术语和概念，并通过一些图示说明了组成一个简单操作系统的典型模块。最后，我们在第 1.5 节简要回顾了操作系统的发展历程，并在第 1.6 节对本章内容进行了总结。

3

1.1 引言

许多年以来，除了操作系统编程人员和计算机迷以外，大多数人对操作系统并不感兴趣。近几年，由于数起备受瞩目的事件，“操作系统”偶尔也会出现在头版新闻中。突然间，就有人认为操作系统可以控制所有计算。人们开始对操作系统的好坏评头论足，甚至出现了非常强烈的不同意见，同时也对操作系统应当提供哪些功能产生了相当大的分歧。许多人（包括一些法院）认为某家公司支配着操作系统市场，而还有一些人则宣称操作系统正逐渐变得不再重要，因为他们认为**互联网浏览器**（Internet browser）就是操作系统。但事实上，操作系统的范畴已变得非常宽泛，类型也日渐多样化，它们存在于每一部可以想到的（包括一些可能使许多人吃惊的）计算设备的某个层级上。

例如，手持式个人数字助理（Personal Digital Assistant, PDA）或称为掌上电脑就拥有功能强大的、复杂但灵活的操作系统。而且，大多数智能型电子设备都拥有复杂但简单易用的操作系统和系统软件来控制自身。一度被认为是神秘的进程管理和内存管理技术的操作系统，现在偶尔也会成为咖啡馆、酒吧或计算机商店的一个话题。当下，许多人看起来都是专家，或者至少对操作

可找到操作系统的一些（也许）令人惊讶的地方：

掌上电脑

有线电视机顶盒

电子游戏机

复印机

传真机

遥控器

手机

汽车发动机

数码相机

系统有自己的观点。

尽管我们也有自己的看法，不过，为了能够更好地解释实际的系统，我们尝试摆脱自以为是的市场营销人员以及数以百万计的用户的喧嚣，深入背后去寻找真谛。当然，一旦需要，我们也会抛出自己的看法，并且说明我们持有相应观点的理由。我们通过列举许多现在使用的系统实例来展示有关概念，并就五花八门的系统，进一步说明什么才是好的而什么才是差的。我们努力回避较真的议题，譬如：“Windows 操作系统与 Mac 操作系统相比，哪个操作系统更好？”或“是否 UNIX 和它的变种（如 Linux）都要强于上述两种操作系统？”相反，我们重点讨论这些系统是如何产生的以及它们为用户和编程人员提供了什么。

4

伴随时代的进步，操作系统的特定部分——特别是那些处理用户和应用程序间交互的部分——被越来越多的用户和编程人员看透或搞清楚，并成为计算机（或电子设备甚至机械设备）销售时的关键因素。买主们正变得越来越挑剔，对操作系统应当提供的品质也寄予更高的期望。同以往任何时候相比，不仅要求系统必须在新功能和简单易用方面提供更多支持，同时还要求系统必须支持那些我们已经习惯了的许多旧的功能和应用程序。理所当然，当我们增加了诸如视频设备和磁盘、高保真音响以及无线网络等新设备时，我们希望系统能够很容易地适应和处理这些设备。事实上，一种好的操作系统体系结构甚至应该支持操作系统构建时尚未出现的甚至可能还没有想到的新设备的连接！

1.2 什么是操作系统

在本节，我们将通过一个简单的例子（即简单的手持式游戏系统）来具体说明操作系统应当提供的一些基本功能。

设想一个非常便宜，但是配备有一个小型显示屏、几个按键和若干游戏的那种手持式电子游戏系统。虽然这类游戏系统可能并不一定需要操作系统，但是，它也可能拥有操作系统，其主要理由在于可以把安装在游戏系统上的各种不同游戏所需要的公共功能模块整合到一起和统一管理。

通常，游戏都有一些共同的模块。例如，每款游戏都需要通过按键来获取输入，也需要在屏幕上显示什么东西。尽管这些动作听起来很容易，不过，它们确实需要一些并不那么简单的软件编程。从一个按键获取相应的输入，听起来简单极了。那么，要是用户一次同时按下两个按键，会发生什么？或者说应该如何处理呢？另外，一款廉价游戏有可能并不使用复杂、昂贵的按键，为此，就可能会发生扭曲传入信号的电子噪声。游戏系统又该如何对此进行处理呢？简单的解决方案是，对每一项这样的公共问题均分别放在唯一的某处地方进行处理。例如，可以把所有按键操作的读取、噪声清理等放在一个独立的软件例程中。设立唯一的 `read-the-button` 软件例程，具有提供一致性用户接口的优势，也就是说，所有游戏均以相同的方式处理按键输入操作。同时，相应例程仅需占用系统内存中唯一的一片空间，而不是在每款游戏中都占有空间。进一步问，`read-the-button` 软件例程应当被放到什么地方呢？答案是，应当放到操作系统中。这样，每款需要读取按键的游戏就都能够调用到该例程。

操作系统还应当处理异常事件。例如，一个用户可能在玩游戏过程中退出游戏（当失败时），并启动另外一场游戏。重启游戏系统不应该是必需的，用户在游戏（任务）之间进行切换是非常自然和预料中的事。事实上，（5岁的）用户可能在不经意的时候按下按键，而当游戏正在进行的时候（甚至当等待按键按下时候），屏幕应当持续被更新（刷新）。这称之为异步性（asynchronicity），通俗而言，就是指事件随机发生或在不可预知的时间发生。即便

像手持式游戏之类的简单系统，也体现出异步性这一非常重要的特征。

5

一些重要的操作系统概念已经融入该游戏系统中：当一场游戏被启动时，其部分软件模块可能被加载进内存，而其他部分则可能已经被提前预装到了只读存储器（Read-Only Memory, ROM）或固定存储器^①中；动态内存被保留下来，以备游戏使用，并且进行了初始化；定时器则可能进行了设定。所有这些都出现在了一款尽管便宜但却有趣的游戏上！你对操作系统还有什么其他的期待呢？

1.3 操作系统的用户视图和系统视图

你或许听说过那句古老的谚语：“每个问题都有两面性。”（或许准确地讲，应当是“两面性或多面性”。）其大意是，就某些问题来说，如果试着从不同的角度去观察，往往有助于我们更好地理解这些问题。为此，从不同的角度来观察和审视事物成为学习新东西的重要方法之一。对于操作系统而言，两种最重要的透视图就是用户视图（user view）和系统视图（system view）。

用户视图属于用户或程序（注意，程序是操作系统的主要使用者）如何利用操作系统的范畴。例如，程序是如何（通过操作系统来）读取按键操作信息的。系统视图则属于操作系统软件如何实施完成相关要求动作的范畴。拿刚才的例子来讲，就是说操作系统是如何获取按键操作信息，进而分离出特定的按键操作（如 shift 按键操作）并使之能够被用户或程序所接受的。贯穿整部教材，我们将同时从用户和系统两方面来阐明操作系统的相关机制、概念和技术。接下来，我们首先详细阐述不同类型的用户及他们对操作系统的认识。

1.3.1 用户视图及用户分类

用户一词经常太过含糊不清，尤其是对于那些在计算中扮演重要角色的人。所以，首先把各种不同类型的用户描述清楚就显得格外重要。然而，尝试确定操作系统的某个用户的角色并不那么简单，因为存在五花八门、各式各样的用户。在此，我们主要想把终端用户、应用程序员、系统程序员和系统管理员区分开来。表 1-1 就上述 4 种主要类型的用户各自分别关切的最重要问题进行了简明扼要的说明。当然，这些问题之间难免会有一些重叠。我们只是试图把有些时候发生分歧的那些观点客观地展现出来。更为复杂的情况是，有时用户可以同时归到若干角色类型甚至所有角色类型上。而这样的用户往往会发现自己拥有相互矛盾或冲突的需求。

应用型用户（application user）或称作**终端用户（end user）**——这类用户包括全部即所有使用（或运行）应用程序或系统程序的人。当我们使用字处理软件、网络浏览器、电子邮件系统或多媒体播放器时，我们是相应的应用程序的使用者。作为用户，我们期望系统（对按键或鼠标移动）能够做出快捷、可靠的响应，提供一致的用户视图（也就是说，能够对诸如屏幕滚动、应用程序退出等各种类型的命令分别采取按类统一的类似执行方式），以及依赖于各种特定类型的操作系统的其他特征。我们在表 1-1 中列出了其他的需求内容。总的来说，这类用户时常只是被简单地称作用户，有时也被称作终端用户。

应用程序员（application programmer）——这类用户包括编写字处理软件或电子邮件系统等各种应用程序的人。一般而言，编程人员对操作系统的要求往往比较苛刻。当他们学

① 我们将会在第 2 章和第 3 章中定义这些术语。

习使用一种新的操作系统时，经常会提及“我如何对一个文件进行读写？”、“我如何接收到用户的按键操作信息？”以及“我如何显示这个方框？”等诸如此类的问题。操作系统提供的实施机制就是程序设计人员的操作系统视图。有时它们被称作系统调用或应用程序接口 (Application Program Interface, API)。它们也可能呈现为编程语言库函数，有时或者就是类包。程序设计人员还要求他们开发的软件可以简单方便地移植到其他平台上。

6

系统程序员 (system programmer)——指那些编写与操作系统紧密相连的软件或模块的人。用于显示计算机网络连接状态的实用例程或可安装的硬件驱动程序均为系统程序的具体例子。系统程序员需要详细了解操作系统的内部功能机制。许多情况下，系统程序需要访问特殊的操作系统数据结构或特权性质的系统调用。尽管操作系统设计人员有时也会关心移植到其他平台的需求，但他们通常对此并不十分关注，他们承担的往往是针对特定平台开发特定功能集合的任务，故而不是很关心可移植性。

系统管理员 (system administrator)——这类用户包括管理计算机设施，因而负责安装和升级操作系统以及其他系统程序与实用例程的人员。他们也负责用户账户的创建和管理以及系统的保护。他们需要详细了解像“如何安装和升级操作系统”“操作系统与其他程序和实用例程之间如何进行交互”等具体事项。为了有效地保护系统和用户，他们也必须了解操作系统的安全和授权功能。

表 1-1 不同类型用户的关注取向

终端用户	简单易用且方便上手 适应用户操作习惯 实时响应输入操作 提供丰富的可视化提示信息 避免令人不快的意外事件（譬如，在没有警告的前提下删除了一个文件） 采取统一的方式处理相同的事情（譬如，在不同的场景中移动一个图标或向下翻卷一个窗口，则移动图标的处理基本一致，翻卷窗口的处理亦基本一致） 为完成同一件事提供其他备选方案（譬如，有的用户喜欢使用鼠标，而有的用户则喜欢使用键盘）
应用程序员	方便程序访问低级的操作系统调用（譬如，读取按键、绘制屏幕、获取鼠标位置） 提供整个系统的一致性的程序员视图 方便使用高级的操作系统机制和服务（譬如，创建新的窗口、读取网络信息、写入网络信息） 针对其他平台的可移植性
系统程序员	方便创建正确的程序 方便调试确定程序问题 方便维护程序 方便扩展现有程序
系统管理员	方便添加或移除诸如磁盘、扫描仪、多媒体配件、网络连接部件等设备 提供操作系统安全服务来保护用户、系统和数据文件 方便升级到新的操作系统版本 方便创建和管理用户账户 平均响应时间表现不错且可预期 系统价格可以负担得起

7

1.3.2 系统视图

系统视图是指操作系统如何提供服务。换句话说，其涉及操作系统的内部工作机制。这是一个不太常见的视图。通常只有一少部分人，即操作系统的设计者和实现者，真正了解或关心操作系统的内部工作机理。事实上，相关信息往往被生产和销售操作系统的公司视作商

业机密。有时，文件管理、程序运行或内存处理等系统主要组成部分的总体工作机理可能被描述和呈现出来，以帮助程序员更好地理解相关子系统的使用。某些情况下，整个操作系统的源程序代码甚至可以被获取到。这样的系统被称作**开源系统**（open source system）^①。

本书的大部分内容主要关注如何这一性质，即系统如何运行一个程序、如何创建一个文件、如何显示一幅图形，等等。为了能够解释清楚真正的“如何”问题的内部细节，我们在书中就实现操作系统功能的若干算法及可选方法进行了介绍和描述。下面，我们将通过跟踪鼠标和光标的移动及管理文件操作两个例子，来具体说明系统视图（或视图）。虽然这些例子看起来可能有点复杂，但它们确实可以说明操作系统是如何真正参与计算机用户所执行的所有操作活动的。

1.3.3 一个例子：移动鼠标（和鼠标指针）

尽管通过移动鼠标（或其他某类**指针式设备**，譬如触控板或轨迹球）来移动屏幕上的鼠标指针（或光标（cursor））的过程看起来直截了当，然而其却可以例证出操作系统的许多视图。图 1-1 具体说明这一过程。当指针式设备被移动时，将触发一种由操作系统负责处理的所谓**中断**（interrupt）的硬件事件。操作系统根据某种硬件专用的单位（即产生的脉冲数，而非毫米或英寸数）来记录鼠标的移动。这是一种**低级系统视图**（low-level system view）。读取鼠标移动的实际软件是操作系统的组成部分，称为**鼠标驱动程序**（mouse device driver）。该设备驱动程序读取低级别的鼠标移动信息，并由操作系统的其他模块对相关信息进行解释和转换为**高级系统视图**（higher-level system view），譬如反映鼠标移动的屏幕坐标。

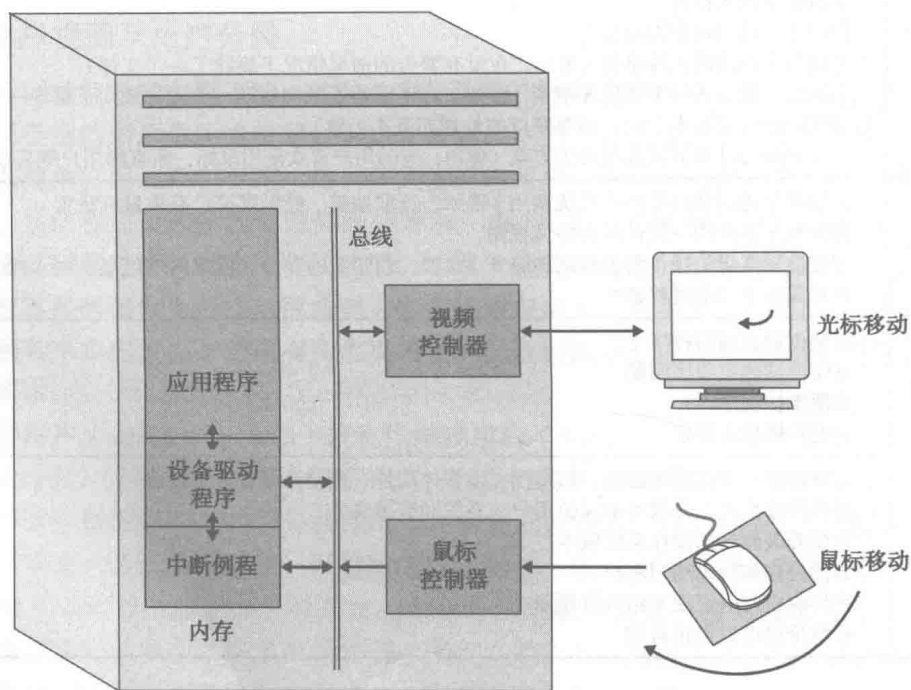


图 1-1 光标对鼠标移动的跟踪

另一方面，或在其他的视图上，是关于“用户看到的是什么？”的问题。在用户视图中，

① Linux 操作系统是众所周知的开源操作系统的一个例子。

光标将在屏幕上快速地移动，而且，如果鼠标以较快速度移动较远距离，那么其在屏幕上的移动看起来也会加快很多。在这些视图之间是**应用程序员视图**（application programmers' view），即“在我的应用程序里，如何获取鼠标移动信息就可以利用它和显示它？”另外一个问题是，“鼠标移动的信息如何提交给应用程序员？”这属于前面提到的高级系统视图的范畴。

让我们结束这些视图的讨论，重新回到系统视图上来。如果存在多个打开的窗口，则哪个应用程序应当接收当前的鼠标移动操作？进一步说，如果在应用程序获取相关操作信息之前发生过多次鼠标移动操作，则这些移动操作可能需要进行排队处理。还有，如果操作系统正在忙于处理其他事务（例如，通过网络连接加载网页），故而无法及时地接收设备驱动输入，那么鼠标移动信息有可能丢失。

1.3.4 另一个比较大的例子：文件

有时，操作系统的最重要的**终端用户视图**是文件系统，尤其是文件名。文件名能包含空格吗？它们可以有多长？是否支持大小写字母？它们按不同的字符还是相同的字符进行处理？非英文字符或标点呢？一种操作系统甚至可能仅仅因为不能使用长文件名或不区分大小写字母就被评判为好或者坏。

在**应用程序员视图**中，文件系统是频繁使用的关键系统部分，其提供新文件创建、现有文件使用、文件数据读、添加数据到文件以及其他文件操作等相关指令。系统甚至可能提供若干不同类型的文件支持。鉴于文件系统的**系统视图**比较庞大，所以其通常被划分为如下几个部分：文件命名和名字处理（目录服务）、文件定位及映射文件名到对应数据等文件服务（文件分配和存储）、尝试把打开文件的部分数据保存在内存中以提高数据访问速度（文件缓冲和高速缓存）、存储设备的实际管理（磁盘调度）。

例如，假定某用户要输入文件名以把相应文件从光盘复制到硬盘。程序可能首先需要检查该文件在光盘上是否存在以及其可否覆盖硬盘上对应文件名的文件。然后，操作系统需要在硬盘的目录中为该文件创建一个目录项；在硬盘上查找空间用于存放数据；从光盘上查找和获取待复制的数据片断（扇区）。而且，所有这一切应当在几秒甚至不到一秒的时间内全部完成！参见表 1-2 所示。

表 1-2 从光盘复制文件到硬盘的步骤

检查该文件在光盘上是否存在
检查硬盘上对应文件——确认可以覆盖
在硬盘的目录中创建对应该文件名的目录项
在硬盘上查找空间以存放文件
从光盘上读取数据扇区
写数据到硬盘扇区
更新硬盘目录
更新硬盘空间信息
在几秒（甚至更少）的时间内全部完成这些操作

8
9

1.4 操作系统的一些术语、基本概念和图解

接下来，我们首先列举和定义一些重要的操作系统概念和术语。然后，再通过一些图示来进一步阐明这些概念。

1.4.1 基本术语

操作系统（或简称为**系统**）：虽然我们能够基于操作系统的不同视图给出其不同的定义，但是下面的非正式定义将是一个很好的起点：操作系统是管理与控制计算机资源或者其他计算设备或电子设备，并为用户和程序使用这些资源提供接口的一个或多个软件模块组成的集

合。这些被管理的资源包括内存、处理器、文件、输入或输出设备，等等。

10

设备 (device): 设备是指与计算机主机系统相连接的硬件。硬盘、光驱和视频监视器是操作系统管理的典型设备。许多设备需要专用的电子 (硬件) 接口, 称为**设备控制器 (device controller)**, 用于辅助设备或一组相似设备连接到计算机系统上。相关例子包括硬盘控制器和视频监视器控制器, 其中, 硬盘控制器拥有许多类型, 并通常遵循如小型计算机系统接口 (Small Computer System Interface, SCSI)、串行高级技术连接 (Serial Advanced Technology Attachment, SATA, 或串行 ATA) 接口以及其他常见但用神秘缩写字母序列命名的行业标准。此外, 作为把设备连接到主机系统的硬件黏合部件, 设备控制器与主机系统之间通常以**总线 (bus)** 相连接。

设备驱动程序 (device driver): 设备驱动程序是组成操作系统的软件例程, 用于实现设备 (经由相应设备控制器) 的通信和控制。

内核 (kernel): 该术语常常指操作系统中实现基本功能且常驻内存的组成部分。某些情况下, 整个操作系统被构建为一个整体并被统称为内核。

服务 (service): 服务是指操作系统内核为用户提供的功能, 大部分采用应用程序编程接口 (Application Programming Interface, API) 或操作系统调用方式。这些服务根据对应的功能可以方便地分组归类, 例如, 文件操作服务 (创建、读、复制)、内存分配服务 (申请、释放) 或其他杂项服务 (获取系统时间) 等。程序设计人员了解和通晓一个系统的关键就在于搞清楚其所提供的有关操作系统服务。

实用例程 (utility): 这些程序不属于操作系统的核心部分 (即内核), 但与内核密切协作, 从而可以简化系统信息的使用或访问。**外壳程序 (shell)** 或**命令解释器 (command interpreter)** 属于实用例程。外壳程序提供了用户访问许多系统服务的接口。例如, 诸如显示一个目录的文件列表、运行一个程序或退出 (注销), 均可能由外壳程序负责处理。外壳程序也可以调用其他实用例程具体完成特定功能。例如, 有些时候, 目录文件列表显示自身就是一个实用例程。

1.4.2 这些图片说明了什么

图 1-2 是一台小型的个人计算机的简化示意图, 反映了一些基本的设备同计算机内存和处理器之间的连接关系。操作系统程序 (或内核) 会包括许多不同的设备驱动程序, 用于负责管理处理器控制下的系统外围设备。例如, 部分内存中存放的数据内容可能会被传输给视频控制器, 进而被显示在监视器上; 而磁盘 (扇区) 的部分内容也可能会被传输给磁盘控制器, 并最终交付给内存 (对于磁盘读操作而言)。

图 1-3 是操作系统的简易示意图。操作系统控制 (或管理) 系统资源, 包括: 控制磁盘、键盘、视频监视器及其他设备; 通过决定哪一个程序启动运行来控制对内存的分配和对处理器的使用; 采用系统调用方式为外壳程序及其他程序提供服务; 通过把硬件设备的复杂细节隐藏起来, 以实现与程序之间的隔离, 从而提供了硬件抽象。

11

图 1-3 经常用来描述操作系统, 但它只是逻辑视图, 而非物理视图。例如, 物理上而言, 操作系统内核驻留在内存单元并运行 (或执行) 在处理器上。故而, 内核 (本身是软件) 与设备 (本身是硬件) 之间的箭头表示逻辑控制, 而非物理控制。

图 1-4 为操作系统的分层视图。最外层代表实用例程 / 应用程序层, 它们访问操作系统内核层, 而后者反过来又管理对硬件层的访问。

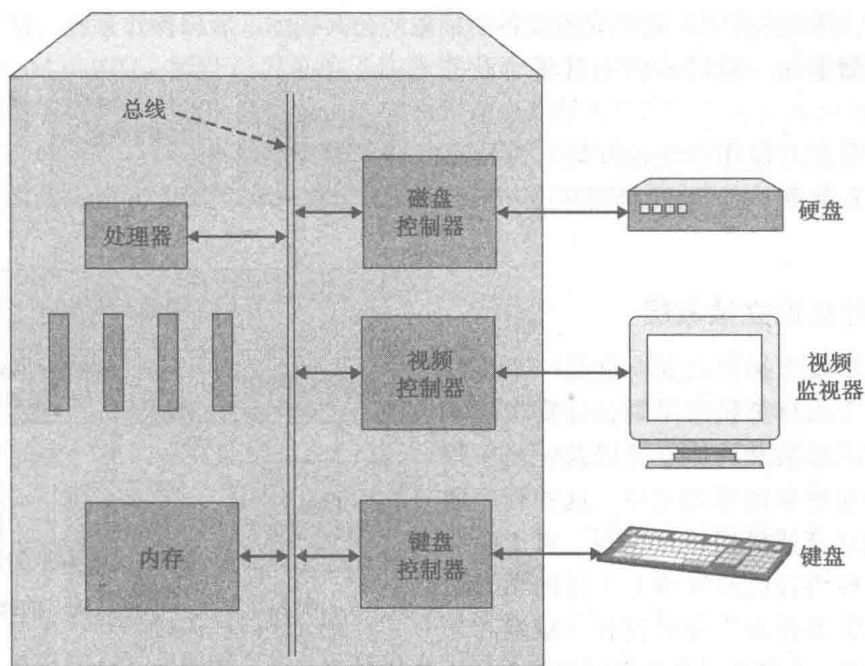


图 1-2 硬件：一台小型的个人计算机的简化示意图

注：这幅图太过简单了。实际上，常常有多条总线，比如说视频与内存之间。我们将在附录中给出更详细的图示。

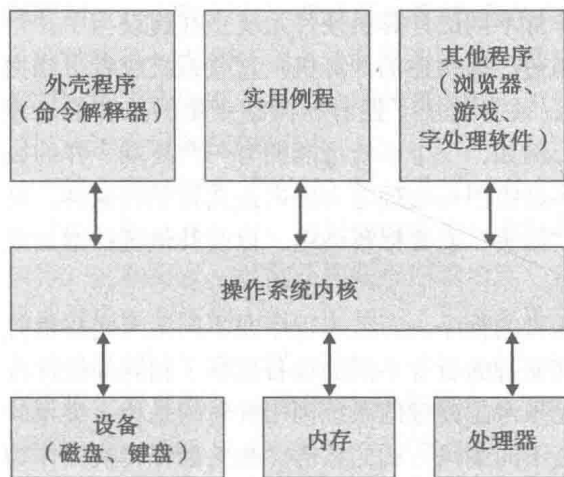


图 1-3 操作系统软件与硬件间关系的简易示意图

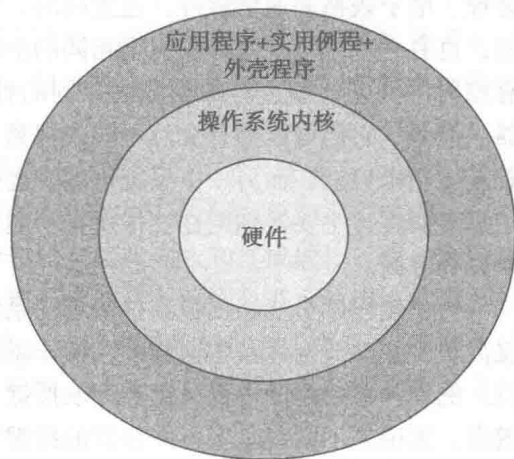


图 1-4 操作系统的分层视图

1.4.3 走近真实：个人计算机操作系统

图 1-5 展示出一台个人计算机（Personal Computer, PC）的操作系统的更多的细节。该操作系统拥有两项在图 1-3 中没有标出的其他模块，即设备驱动程序和基本输入 / 输出系统（Basic Input/Output System, BIOS）。基本输入 / 输出系统对硬件进行了抽象。换句话说，基本输入 / 输出系统负责管理如键盘、基本视频和系统时钟等常见设备。这样做可以支持操作系统主体或高级部分采用相同的方式来处理相同类型的设备（如所有的键盘）。于是，无论键盘有 88 个按键、112 个按键还是其他数量的按键，甚至对于由于不同语言字符或重音键而可能出现在不同键盘上的按键没有出现的情况，操作系统内核都无须发生改变。设备驱动程序也对相似的设备提供了类似的抽象。例如，数字化视频光盘的设备驱动程序可以由设备

制造商提供, 并对操作系统提供光盘设备的抽象或公共视图, 所以操作系统可以不考虑制造商, 也无须跟着每一款特殊的光盘驱动设备而发生改变。

下一节将就“操作系统设计时, 为什么设立抽象层非常重要?”的问题展开进一步的深入探讨。

13

1.4.4 为什么设立抽象层

为什么要设立抽象层呢? 这是一个很好的问题。在个人计算机的早期, 计算机玩家一度沉浸在因组装和构建硬件以及让简单程序工作起来而带来的乐趣之中。这些程序通常采用汇编语言或机器语言编写。对于某些人来讲, 这些语言已经够得上不错的学习工具, 然而相关编程确实非常冗长、枯燥和乏味。

即便如此, 人们希望享受编写更有趣因而更长、更复杂的程序的经验。于是, 更好的工具不可或缺。这样的工具包括简单易用的编辑器和面向特定高级程序设计语言的编译器或解释器。对于终端用户而言, 则渴望把计算机当作商业或生产工具来使用。这些用户需要文字处理、电子表格和通信软件。与此同时, 许多非常不同的计算机硬件系统正在建设当中。当然, 也有一些相似的但并非完全相同的由诸多制造厂商构建的计算机。这些系统或者可能拥有相同的处理器制造厂商制造的相同的处理器, 或者使用了拥有相同指令集的兼容的处理器。但是, 它们可能拥有全然不同的视频设备。例如, 一个系统可能拥有一台终端一样的设备连接在串口上, 而另一个系统则拥有提供许多高级图形处理能力的内嵌式视频控制器。对于键盘而言, 主要区别则往往体现在功能键或“箭头”或光标移动键, 以及其他键的增加或缺失等方面。

为了使程序员能够创建在这些不同系统上运行的程序, 而且当程序在不同系统间移植时仅需很少变动甚至无须任何修改, 操作系统为其支持的所有不同的设备提供了相同的硬件接口。例如, 一个程序可以从键盘读取按键, 而无须考虑读键的系统调用针对的是何种类型的键盘。无论是不同键盘上不同位置的按键, 还是不同编码方式的按键, 相关翻译转换工作均由操作系统负责处理完成。

为避免因使用不同键盘、不同视频监视器、不同磁盘等就需要采用不同版本的操作系统可能带来的复杂性和成本暴涨, 操作系统被划分为适应不同硬件设备的设备相关部分(包括基本输入/输出系统和设备驱动程序)以及面向所有硬件保持相同的公共部分(见图 1-5 的内核)。这种将复杂工作分成若干层或层级的技术, 是广泛应用于包括操作系统在内的大型复杂软件系统的成熟的软件技术。这样, 维护操作系统适应新的兼容性计算机系统及不同设备, 就主要涉及基本输入/输出系统和设备驱动程序的修改(或重写), 而其余内核模块、程序和实用例程均可以保持基本不变。这对任何人来讲, 无论用户、制造商还是操作系统开发人员, 都是一个非常有吸引力的想法。

如果一家计算机外围设备制造商(例如, 视频卡制造商)设计了一款新的设备, 并希望把它卖给用户, 这样他们就可以把计算机升级到新的硬件设计方案上来。这时, 新的问题又

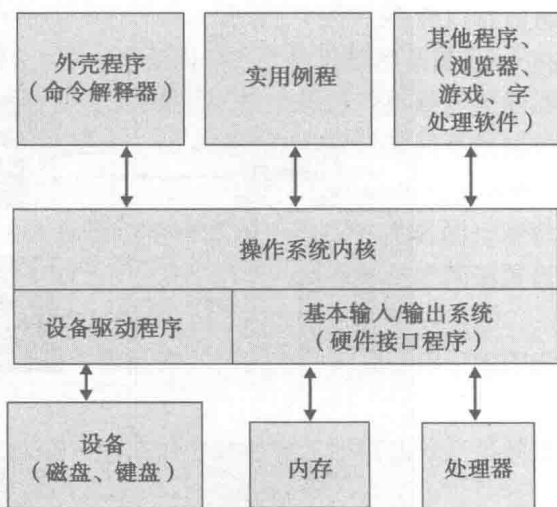


图 1-5 个人计算机（小型系统）的操作系统模型

出现了。因为，现有计算机中的基本输入/输出系统通常是安装在只读存储器（Read Only Memory, ROM）中的，所以相关更换工作非常困难且成本昂贵。解决这个问题的方法是，在操作系统被加载到内存时，允许设备驱动程序动态加载和创建一个可修改的基本输入/输出系统。系统启动时允许基本输入/输出系统的代码发生替换，将可以方便支持计算机添加新的功能，或者用新软件替代基本输入/输出系统的现有功能，或者支持在现有硬件上扩展新的功能。

14

1.5 操作系统发展导论

在本章即将结束之际，我们先从历史的视角出发来简单回顾一下操作系统是如何发展起来的，然后讨论关于操作系统应当包括的功能类型的不同看法。关于操作系统发展的更详细的历史时间表，我们将在介绍过一些其他概念的基础上，于第3章结束时展开进一步的说明。

1.5.1 操作系统的起源

当然，在个人计算机出现之前，曾经有过许多较大型的计算机。最初，这些机器是非常庞大、非常昂贵的。然而按照现代的标准来看，它们却是非常落后和原始的，而且几乎没有什么程序设计人员。由于主存容量非常小、中央处理器（Central Processing Unit, CPU）运算速度非常慢且只有一些简单的输入/输出设备，所以程序在处理能力方面受到极大的制约和限制。一台典型的早期计算机系统可能拥有仅仅几千字[⊖]大小的主存、每秒执行数千条指令的处理器以及用作输入/输出设备的一部电传打字机[⊗]。鉴于这些早期计算机的处理能力非常有限，所以那些大部分用计算机基本机器代码、机器语言或汇编语言编写的程序必须经过仔细思量和精雕细琢。

这些程序非常了不起！短短几百或几千条机器指令，却完成了让人吃惊的大量的工作。不过，它们都面临着一些相似的需求：程序如何打印输出结果？程序如何加载到内存并开始执行？这些需求，即需要加载程序到内存、运行程序以及接收输入和生成输出，就成为构建早期的操作系统的动力。起初，工作在系统上的为数不多的几个程序员彼此认识，常常相互分享已经调试过的例程（程序代码）以简化编程任务。这些被共享的例程（譬如，“打印电传打字机上寄存器A的取值”）最终被合并和形成一个库（library），该库可以与应用程序合并（链接）到一起，从而构成一个完整的可执行的程序。

这些早期的计算机是单用户系统（single-user system），也就是说，任何时候都只能有一个用户和一个程序在运行。通常，程序员会按较小的时间段（或许是按10~15min的时间段递增处理）来预订对计算机的使用。程序员将使用这段时间来运行或调试一个程序。由于计算机非常昂贵，而计算机的时间又非常宝贵，所以大块可用的时间段往往只出现在夜间或清晨。那个时候，万籁俱静，周围几乎没有什么管理人员，程序员可以比在白天做更多的事情。这一始于计算早期的传统，成为一直持续到今天的少数几项传统之一！

程序一旦编写完成和经过汇编处理，进而与输入、输出、数学函数[⊕]、打印结果格式化

15

⊖ 一个字通常为6字符，但不同系统会有区别。

⊗ 电传打字机由机电式打印机和用于收发报文的键盘组成，能以每秒十几个字符的速度打印输出或打字输入。

⊕ 早期的计算机硬件通常没有复杂的数学甚至算术操作（譬如长除法）的指令，所以这些操作均以软件实用例程的方式来实现。

处理以及其他常见任务等实用例程链接或绑定到一起，形成一个可执行程序，就可以加载到内存中并启动运行。该程序可以存储在穿孔的纸带上或穿孔卡片上。计算机硬件将知道如何从输入设备开始读取，但它只加载第一张卡片或纸带的第一块区域。因此，那里必须要包括一个小的例程，以用于把应用程序的其余部分加载到内存中。这个简短的例程被称为装入程序（loader）或称为加载程序。装入程序将依次读取程序员的可执行程序，并把它和所需的实用例程存储在内存中一个指定的位置，通常是第一个内存地址或某个特殊的固定位置。然后它将通过一个分支或“子程序”调用来转向执行由它所加载的程序。被加载的程序纸带或卡片叠的样子如图 1-6 所示^①。结束（END）定界符用于提示装入程序，已无其他例程需要加载，因为例程之后可能会紧跟有数据记录。



图 1-6 带有装入程序和类似于操作系统的实用例程的应用程序

时间一长，程序员开发出越来越多的实用例程，装入程序也因而变得更加复杂。装入程序很快能够加载由更高级别编程语言翻译（编译）得到的程序。伴随装入程序、实用例程及用户程序的大小的增长，卡片叠或纸带变得越来越大（为此，诸如卡片叠丢失、纸带撕裂等不幸事件也变得司空见惯）。这些装入程序和实用例程成为早期操作系统的开端，并通常被称为监控程序（monitor）。

1.5.2 操作系统应当做什么

从计算的早期一直到今天，一场关于“操作系统应当做什么？”的激烈辩论——从斯文的讨论到政治式或几乎宗教式的争论——就从来没有停息过。这场辩论的两种极端观点可以被称作最大化观点和最小化观点。最大化观点认为操作系统应当囊括尽可能多的功能，而最小化观点则认为只有最基本的功能才应当作为操作系统的组成部分。从早期的系统起，就已开始提出这样的问题：“所有关于输入/输出的例程都应当包含在我的程序中吗？我甚至都不从读卡器读东西呀。”包含太多的例程——任何不必要的例程——让程序的可利用内存空间变得越来越小，甚至由于太小而无法启动运行。如何才能得到刚好需要的东西呢？诸如执行浮点运算的数学例程可以在操作系统中只包含一次，而不用单独包含在每个用户的程序中。但是，这样一来，每个程序都必须承受由操作系统中包含这些例程占用内存所带来的额外开销，即便是对于没有使用浮点运算的那些程序（如会计应用程序）也是如此。

[16]

直到最近，关于“什么应当包含在操作系统中？”的辩论仍在继续。例如，现在普遍认为，用户友好的操作系统接口应当包括指针式设备（如鼠标、轨迹球或触控板）及某种类型的带有下拉式菜单的屏幕窗口。所以，此类接口是应当作为操作系统的一部分（故而赋予所有的应用程序类似的“外观和感觉”），还是作为外壳程序的一部分（以方便让每个用户自己决定他们想要的特定的外观），就成为当前关于“操作系统应该包括什么？”的辩论议题之一。

公平地说，就像许多热议的话题一样，最大化观点和最小化观点都有自己的道理。就目前来看，历史发展的趋势尚不明朗。对于较新的操作系统而言，在某些情况下已变得更小、

① 这种类型的装入程序通常被称为引导装入程序（bootstrap loader）。

更简单且可配置程度更高，而另外一些情况下则恰恰相反，操作系统变得更大、功能更强、制约更多。关于“什么功能应当安排在什么地方（操作系统内核还是其他位置）？”的问题导致了各种不同的操作系统设计的可能性，对此我们将在第2章展开进一步的讨论。

1.6 小结

本章中，我们首先介绍了操作系统的一些基本功能。我们给出了几个简单的例子，具体说明了为什么操作系统那么重要。然后，我们从观察操作系统的两个角度（即用户视图和系统视图）出发，围绕“操作系统做什么？”的问题，讨论了各种不同的观点。在此基础上，我们提出了一些基本的术语和概念，并通过一些图示阐明了简单操作系统的典型模块组成。接下来，我们介绍了实际构建操作系统常用的几种体系结构，并讨论对于操作系统成功设计特别重要的“抽象”概念所特有的思想和理念。最后，我们从历史角度出发，简短回顾了操作系统的起源。

下一章将概括介绍操作系统的主要模块，并进一步详细讨论系统结构的选择。

习题

- 1.1 用一句话来给出操作系统的定义。
- 1.2 既然我们中的绝大多数人将来都不从事操作系统的编码，为什么我们还需要学习操作系统的相关知识呢？
- 1.3 为什么如手持式电子游戏机这样的简单设备都可能包含有操作系统？请给出3个理由。
- 1.4 操作系统的用户视图与系统视图的主要区别是什么？
- 1.5 教材就哪4种不同类型的用户进行了讨论？他们大都对操作系统的什么方面感兴趣呢？
- 1.6 本章从系统视图出发，讨论了不同用户是如何被支持的。其间，列举了关于鼠标移动和文件系统两个例子。考虑操作系统的另外一个方面，讨论系统视图是如何支持3种不同类型的用户的。
- 1.7 操作系统是应当被专利化，从而使制造商可以赚取足够利润来继续他们的开发，还是应当把操作系统的核心部分和规范说明开放给所有用户，让大家都知道？[⊖]
- 1.8 就操作系统的学习而言，控制器如何来定义最好？
- 1.9 抽象的总原则是什么？
- 1.10 为什么我们要对操作系统进行抽象？有些什么理由呢？
- 1.11 如何区分操作系统和内核？
- 1.12 简明扼要阐述早期大型主机系统上操作系统的起源。
- 1.13 窗口界面特征（即决定外观和感觉的相关因素）是应当作为操作系统内核的一部分，还是作为命令式外壳程序的一部分？

[⊖] 任课教师请注意：不要把这个问题作为课堂教学内容的一部分，除非那天你没有别的什么可以谈论。

操作系统概念、模块和体系结构

在这一章中，我们讨论操作系统总体上所做的工作，并概要说明操作系统的概念和组成模块，从而使学生对操作系统有一些整体的认识。我们还讨论了一些在几乎所有的操作系统中都会用到的常见技术。

为了能够了解操作系统是如何参与到几乎所有的系统操作中的，我们在第2.1节一开始就给出一个简单的用户场景，并描述了该场景中由操作系统所采取的一些行动。在第2.2节中，我们概述操作系统所管理的系统资源主要类型。这些资源包括处理器、主存、输入/输出设备和文件。然后，我们就主要的操作系统模块及每个模块所提供的服务进行了概括性说明。相关模块包括进程管理和处理器调度模块、内存管理模块、文件系统模块、输入/输出管理和磁盘调度模块。在任何特定的操作系统中，它们可能会被实现为独立的模块，但也可能不会，不过这里把它们独立看待将有助于更方便地解释操作系统的相关概念。

19

接下来，在第2.3节中，我们定义进程的概念（要知道，进程是操作系统所有工作的核心所在），并描述了进程的状态以及由操作系统所维护的关于每个进程的一些信息。一个进程（有时称为作业或任务）^①基本上就是一个正在执行的程序，而操作系统则代表所有的进程统一管理相关的系统资源。在第2.4节中，我们讨论不同类型的操作系统的特点，从一次能够运行或执行一个单个进程的系统，到管理并发执行的进程的系统，再到分时和分布式系统。

在第2.5节中，我们介绍构建操作系统所采取的一些不同的体系结构及方法，具体包括整体式操作系统结构、微内核结构和分层式结构。然后在第2.6节中，我们描述一些被各种操作系统模块经常用到的实现技术。其中包括由多任务操作系统维护的队列，后者用于记录那些等待获取资源或等待特定服务执行完成的作业。例如，进程可能等待磁盘输入/输出、处理器时间或打印服务。我们还描述了中断和如何处理它们的一些细节、面向对象的操作系统设计以及虚拟机。第2.7节围绕“什么功能应当包括在操作系统里？”议题展开一场哲学式的讨论。最后，在第2.8节中，我们对这一章的内容进行总结。

2.1 操作系统做什么工作

在这一节中，我们将通过一个小例子及相应的场景分析来具体说明操作系统是如何参与到计算的几乎所有方面的。请设想和考虑如下简单的用户场景：

某用户想输入一段不长的留言用于提醒自己^②。事情的起因是这样的：今天早上他开始工作时，听到了一则电台广告宣称他最喜欢的音乐组合即将莅临他所在的小镇进行演出，于是，他想提醒自己购买门票和邀请一些朋友。为此，他开启了一个日程管理程序（或可能是文本编辑器或字处理程序），输入自己的提醒内容，保存文档，然后退出。该用户可能使用

① 术语作业和任务在某些文献中用于指相同的概念，但在另外一些文献中则用于指不同的概念。今后遇到此类情况，如有需要，我们将会以脚注方式加以讨论和说明。

② 为简化叙述语法及统一相关代词，本文将假设该用户是一个男性。

了掌上电脑、基于窗口的系统（例如，Mac、微软 Windows 或 Linux 操作系统及基于图形化用户界面的文本编辑器），或者是一个简单的基于文本的命令式外壳程序（如 UNIX）。在此，让我们假设他正在使用一个基于图形化用户界面的文本编辑器来写一段独立的提示信息，并将其保存为一个文件。为完成这样的工作任务，不论使用的是什么类型的系统，相关场景下均会引发操作系统去创建、管理乃至最终终止对应的软件模块。当这个用户启动编辑器或其他某个程序时，他便创建了一个进程（也称为任务或作业）^①。一个进程基本上就是一个执行（execution）中的程序。一个进程可能正等待着要运行、正在运行、正等待着发生某件事情或者完成。进程可能等待的一些事件包括来自用户的按键操作、源自磁盘驱动器的数据读取操作或者读取另外一个程序所提供的数据的操作。

在一个进程能够启动运行之前，必须首先把要运行的可执行程序文件（二进制类型）加载到主存中。该文件通常是从磁盘或某种电子存储器（如闪存驱动器）加载进内存的。期间，操作系统需要参与实施若干主要的活动以完成相关工作任务。首先，需要有一部分主存来存放程序的可执行代码。同时，程序的数据、变量和临时存储机制（译者注：堆或栈）都需要额外的内存。在我们的示例中，数据将是用户在备忘录文件中创建的条目。这些分配内存的活动是操作系统必须要做的内存管理的一部分。通常，可能有若干程序同时存放在内存中。操作系统内存管理器模块负责控制哪些进程被放置在内存中，放置在什么位置，以及每个进程分配得到多大的内存空间。进程管理（process management）则是操作系统的另一个关键管理活动，其决定哪一个进程开始运行、运行多长时间以及（有些情况下）运行在什么优先级（或重要性级别）上，并通常由操作系统的处理器调度程序部分负责处理。

20

一旦编辑器进程处于运行状态，它便需要接受一些按键操作并把输入的内容显示在屏幕上。即使设备是没有键盘的掌上电脑，字符也会以某种方式在操作系统控制下被输入和接受。获取按键和字符，并在屏幕上显示那些字符，相关工作是通过操作系统的输入/输出和设备管理模块经由一系列步骤具体完成的。

当我们的用户点击一个按键时，他便输入了一个必定会被系统读取的字符。事实上，这个设备——当前情况下，其实就是一个键盘——输入的是关于原始的按键操作的信息。相关信息——对应按键在键盘上的位置（行号和列号）以及是否被按下或释放——存储在一个临时缓冲区中。在掌上电脑或个人计算机中，可能会有一块特殊的键盘控制器芯片用于存放按键操作信息，并负责向处理器发送中断信号。除了在键盘上的控制器芯片外，处理器也可能拥有自己的键盘设备控制器。中断会导致处理器停止其当前正运行的进程。如果处理器正进行的是较低优先级别的工作，可能会立即这么做。但如果处理器已经开始和正在进行的是较高优先级别的工作，则可能随后再做。接下来，操作系统会启动一个中断服务例程来处理键盘操作。该中断服务例程是操作系统中中断处理和设备控制的组成部分。对于每个字符的输入，均会重复以上处理过程。相关字符被发送给编辑器进程，并在屏幕上显示出来，则是经由操作系统的另一项活动。就当前这种情况而言，执行的是对视频监视器的输出操作。

当我们的用户完成留言输入操作后，他将他的留言保存为一个文件。这可能会涉及移动一个指针式设备（如鼠标）以指向屏幕上的文件菜单。鼠标的移动和点击首先是由一个设备控制器处理的，后者跟踪鼠标的坐标并将它们发送给操作系统。用于跟踪鼠标的图标（例

① 启动程序有时也被称为实例化、执行、加载或运行程序。

如，箭头）必须同步移动，并在监视器显示屏上显示出来，这是另外一个到屏幕上的输出操作。当鼠标键按下时，对应控制器将相关信息发送给操作系统，后者把鼠标键按下时的坐标信息进一步发送给管理用户界面的窗口系统。窗口系统拥有关于“哪个窗口是当前活跃窗口？”以及该窗口中各类按钮和其他图标的位置的信息。利用这些信息，窗口系统将会把当用户按下鼠标键时的光标的坐标，匹配到被“点击”的特定屏幕按钮图标（或符号）上。一般来说，处理用户交互的窗口系统是相当复杂的。某些人认为窗口系统是一个系统程序（system program），与操作系统相对独立。但其他的一些人则认为，它是操作系统不可分割的一部分（参见第 2.7 节关于“什么是操作系统的组成部分？”的讨论）。

[21]

继续我们的情景分析，我们的用户可能会选择一个名为“个人备忘录”的目录用于存放他的文件。这便引发操作系统的文件管理（file management）模块的运作。当用户选中对应目录（例如，通过双击文件夹图标）时，将导致操作系统的文件管理器采取一系列动作。首先，它必须通过从操作系统的内部表格中检索相关目录信息，打开对应目录。这里，目录信息包括存放在该目录下的文件（也可能是其他目录）的名称，以及该目录在磁盘上存储的位置信息。然后，用户必须输入一个文件名，如“演唱会提醒”，于是，文件系统将检查核对并确保该目录中没有任何现有文件具有相同的名称。进而，磁盘空间分配模块被调用，尝试在磁盘上找到一块合适的空闲空间区域来存储文件。最后，操作系统文件管理器将在该目录中创建一个文件目录项，用以包含新文件的各种信息，如文件名、文件类型和其在磁盘的存放位置等。

正如我们从这个非常简单的例子中可以看到的那样，操作系统参与到了用户和程序交互的几乎每一个方面——从低级别的活动（如处理键盘按键和鼠标移动操作），到资源分配算法（如分配内存空间和处理器时间），再到更高级别的活动（如管理文件名和目录）。关于操作系统如何处理所有这些不同任务的描述和介绍，将贯穿本书的始末。

2.2 操作系统管理的资源及主要的操作系统模块

操作系统的一项主要作用就是对系统资源进行管理。为此，本节介绍操作系统管理的主要资源类型。然后，通过阐明主要的操作系统模块、每个模块所管理的资源以及每个模块所提供的服务和功能，完整地刻画出一个典型的操作系统的概念视图。

2.2.1 操作系统管理的资源类型

本小节首先介绍典型操作系统所管理的主要资源。这些资源包括最底层的处理器、主存、高速缓存、辅助存储器和输入/输出设备，以及较高层级的文件系统和用户界面。另外，操作系统还对网络访问进行管理，并对其所管理的各种资源提供安全机制来实施保护。

处理器

操作系统需要随时调度和确立每一个处理器上运行哪一个进程。在较早的单进程系统中，这非常简单，因为整个内存空间仅有一道进程驻留其中，操作系统主要负责把处理器的控制权交给驻留内存的那道进程，从而使其启动执行即可。然而，即使在这样一个简单的系统中，在让进程控制处理器之前，操作系统也必须完成一些其他的事项，例如，设置内存保护寄存器和切换到用户执行模式等。

在多任务系统中，处理器资源的管理是相当复杂的，因为可能同时有多个进程驻留在内存空间。如果系统中包含有多个处理器，则情况可能会变得更加复杂。操作系统将会维护各

种各样的进程队列。与处理器调度最相关联的队列称为就绪队列 (ready queue)，其中包含有已准备就绪并可执行的所有进程。如果进程拥有不同的优先级，则可能需要为每个优先级别单独设立一个相应的就绪队列。每个进程通常会被设定一个控制处理器的最长时间段，称为**定量时间** (time quantum)，即时间片 (time slice)。如果该定量时间已用完，即便进程尚未执行结束，定时器中断也会即时启动一个称为**上下文切换** (context switching) 的操作系统进程，把处理器控制权切换给另一个进程。我们将在第 9 章中详细讨论处理器调度算法以及操作系统是如何管理处理器资源的。

22

主存和高速缓存

操作系统需要在进程可以执行之前为其分配内存空间。程序的可执行代码通常存储在硬盘 (或其他的辅助存储介质) 上。当用户或程序要执行一个驻留在磁盘的程序时，操作系统必须在磁盘上查找和确定程序代码文件，同时分配足够的内存空间来保存程序的初始部分。由于许多程序都非常大，所以操作系统可能从磁盘只加载程序的一部分到内存。内存管理的主要功能之一就是为进程分配初始的内存空间，并在进程需要时再从磁盘加载程序的其他部分到内存。如果所有的内存空间都被占用，那么操作系统的内存管理模块必须把某些驻留内存的信息**对换** (swap out) 到外存，以便其可以加载进程所需的其他部分到内存。我们在第 10 章和第 11 章讨论内存管理技术。

辅助存储器

操作系统管理的另外一种重要的资源是辅助存储器，通常为硬盘。大多数的程序代码文件和数据文件都存储在硬盘上，除非发生需要把它们加载到内存的请求。每当一个进程所要求的数据或代码不在内存时，就会有一个请求被发送到操作系统的磁盘调度模块。操作系统通常会暂停请求者进程，直到所需数据读入到内存为止。在多任务系统中，可能有许多关于磁盘数据的读取 (加载到内存) 或写出 (存储到磁盘) 的请求。操作系统往往会维护磁盘读写请求的一个或多个队列，并使用各种算法来优化相关请求服务。我们将在第 14 章讨论磁盘调度 (作为输入 / 输出管理讨论的一部分)。

输入 / 输出设备

操作系统还必须管理和控制连接到计算机系统的各种输入和输出设备^①。操作系统包含有被称为**设备驱动程序**的模块，用于实现对这些设备的访问控制。因为现在有许多不同类型的输入 / 输出设备，且用户经常给他们的系统添加新的输入 / 输出设备，所以现代操作系统能够检测到新硬件并动态地安装相应的设备驱动程序。设备驱动程序处理与设备控制器之间的低级别的交互，并为操作系统的其余部分呈现一个关于输入 / 输出设备的较高级别的视图。这样，操作系统就能够以一种抽象和统一的方式来处理相似的设备。我们将在第 12 章中讨论输入 / 输出管理。

23

文件系统

上述讨论的资源都是硬件资源，所以均应归为低级资源。操作系统还管理着通过软件创建的较高级别的资源，其中，一种主要的此类资源是**文件系统**。作为操作系统的模块，文件系统提供了一种较高级别的接口，允许用户和程序对各种类型的文件执行创建、删除、修改、打开、关闭及其他操作。最简单的文件类型就是把文件内容看作一个字节序列。当然，也可以是较复杂的文件结构，例如，将文件内容组织成记录。文件系统允许用户对文件

^① 把磁盘管理作为输入 / 输出管理组成部分的看法并不少见，因为无论磁盘还是输入 / 输出设备，都是输入 (读) 字节到内存，或从内存输出 (写) 字节。

命名，将文件组织到目录中，保护文件，并使用各种文件操作来访问这些文件。我们将在第 12 章对文件管理展开进一步的详细讨论。

用户界面

许多现代操作系统包含有另外一个高级模块来处理与用户之间的交互。其包括在计算机屏幕上创建和管理窗口的功能，从而支持用户与系统进行交互。通过在操作系统中设立这样的一个模块，用户就能够以统一的方式来访问各种各样的资源。例如，访问文件系统上的目录和访问互联网上的文件将通过一个统一的接口来进行处理[⊖]。我们在本书的各个章节都会讨论到用户界面。

网络访问

操作系统管理的另一种资源是网络访问，相关资源管理功能支持一台计算机上的用户和程序访问计算机网络上的其他服务和设备。操作系统既提供低级的网络访问功能，也提供高级的网络访问功能。低级功能的一个例子是支持程序创建网络端口，并连接到另一台机器的端口上。高级功能的一个例子是支持访问远程文件。我们将在第 15 章和第 17 章讨论网络和分布式系统。

提供保护与安全

操作系统还提供了保护各种资源的机制来防止未经授权的访问，同时提供相关安全技术支持系统管理员实施他们的安全策略。最简单的安全类型是通过密码来实施访问授权，但一般来说这是不够的。我们将在第 16 章讨论安全和保护。

2.2.2 操作系统的主要模块

图 2-1 在某抽象级别上对操作系统的一些主要模块进行了说明。毫不奇怪，其中许多模块与其所管理的资源几乎完全对应。另有一些模块提供了诸多其他模块所使用的通用支撑功能。操作系统模块既提供了由系统用户和程序所访问的功能，也提供了由其他操作系统模块所访问的功能。某些功能仅限于由其他操作系统模块在特权模式下进行访问，例如，设备驱动程序的功能往往仅限于操作系统内部进行访问。其他的功能，如文件系统功能，则可由操作系统模块、用户或应用程序所访问。

24



图 2-1 主要的操作系统模块

在图 2-1 中，我们并未标出操作系统模块之间是如何相互作用的。这是因为，相互作用的类型取决于操作系统实现所采用的特定的体系结构。例如，在层次式体系结构（layered

⊖ 就像我们先前提到的那样，用户界面有时被认为是系统程序的组成部分，而非操作系统必不可少的一部分。

architecture) 或称为分层式体系结构中, 相关模块将被划分成若干层级。一般而言, 某个层级的模块可以调用同一层级或较低层级的模块所提供的功能。另一方面, 在面向对象的体系结构 (object-oriented architecture) 中, 每个模块将被实现为一个或多个提供服务的对象, 且任何对象都可以调用其他对象所提供的服务。在整体式体系结构 (monolithic architecture) 中, 所有的模块将被统一实现为一个庞大的程序。我们将在后面一节中讨论最常见的几种操作系统体系结构。

2.3 进程概念和操作系统进程信息

考虑到进程在操作系统概念中的核心地位, 现在我们来介绍进程的概念。首先, 我们将定义进程是什么, 并描述进程可能经历的各种状态以及导致进程发生状态转换的事件类型。接下来, 我们将讨论操作系统为管理进程和资源而必须维护的各种信息。我们还将介绍进程控制块 (Process Control Block, PCB) 的概念, 即操作系统用于跟踪每一进程而维护的数据结构。最后, 我们将对各种不同的进程进行归纳和分类。

2.3.1 进程定义和进程状态

一个进程就是一个正在运行的程序。要想成为一个进程, 程序需要由操作系统启动执行。不过, 一个进程在其存在的整个期间并不一定一直在运行。例如, 它可能会等待输入/输出 (譬如, 某个按键被按下) 或者等待操作系统为其分配某些资源 (譬如, 一块内存)。每个进程都有一个特定的执行序列以及一个用于指定要执行的下条指令位置的程序计数器 (program counter)。它也将拥有操作系统为其分配的各种资源。例如, 它将需要一些内存空间 (memory space) 来存储其所有的或部分的程序代码和数据 (如程序变量)。它将几乎肯定会访问文件, 所以它也可能拥有一些相关联的打开文件 (open file)。进程有时也被称为作业[⊖] (job) 或任务 (task), 我们将交替使用这些术语。

25

一旦进程被创建, 它就可能处于如下某种状态: 运行状态 (若其拥有处理器控制权)、就绪状态 (若所有的处理器都正被其他进程所使用)、等待状态 (若其需要等待某个事件的发生), 等等。一个进程可能经历的典型状态如图 2-2 所示, 称为状态转换图 (state transition diagram)。图 2-2 中的结点 (呈现为六边形) 表示进程状态 (process state), 而有向边 (directed edge, 即箭头) 表示状态转换 (state transition)。现在, 我们来讨论这些状态以及触发状态转换的对应事件[⊖]。

0 号状态转换将创建一个新的进程, 该状态转换可能会因如下某种事件而触发:

1) 一个正在运行的操作系统进程可能创建或克隆一个新的进程。例如, 当一个交互式用户登录到计算机系统中时, 负责登录的操作系统进程通常会创建一个新的进程来处理与用户之间的交互以及其所输入的命令。操作系统也可能会创建一些新的进程 (如中断处理器进程或错误处理器进程) 来负责承担操作系统的某些功能。

2) 一个用户进程也可能通过调用操作系统中创建新进程的函数来创建另一个进程。例如, 网络浏览器可能创建一个新的进程来运行外部的“插件”模块, 从而处理在网站上访问

⊖ 历史上, 术语作业曾指采用所谓作业控制语言 (Job Control Language, JCL) 编写的可以调用各种任务的控制序列, 相关解释主要应用于早期的批处理系统中。

⊖ 该状态图较为典型和常见, 但对于特定的操作系统而言, 操作系统设计者有可能想要分辨其他的状态, 于是人们就会在系统内部或文档中看到更少状态或更多状态的情况。

到的特定类型的多媒体内容。

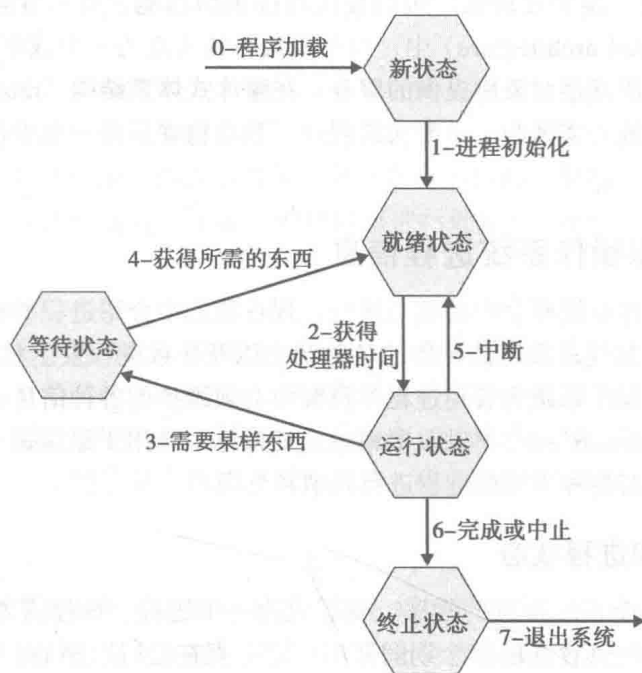


图 2-2 进程状态及转换的简化示意图

3) 当某道作业由操作系统以预定事件方式启动时 (例如, 在 UNIX 系统上的一道 “cron” 作业, 即周期性或例行性执行的作业), 操作系统将创建一个进程来执行该道作业。

当一个新的进程被创建时, 其处于新状态 (new state)。操作系统必须建立用于保存进程有关信息的表格 (见第 2.2.2 节), 分配必要的资源 (如用于存储程序的内存), 查找定位程序可执行文件和该进程所需的任何初始化数据, 并执行适当的例程把进程的初始化部分加载到内存中。图 2-2 中的 1 号状态转换表示操作系统把进程从新状态转换到了就绪状态 (ready state), 后者表明该进程已经准备就绪并具备除处理器之外的所有执行条件。注意在 1 号状态转换可以发生之前, 操作系统必须被允许添加一个新的进程。具体而言, 一些操作系统可能对特定时间允许进程的最多数量进行了限定, 所以如果进程数已经达到最大值, 则将不会准许添加新的进程。在大型主机系统或集群系统中, 还可能存在作业执行之前就必须拥有足够可用资源的要求, 这可能是一个特定的输入 / 输出设备, 也可能是一定数量的处理器。只有当所有这些初始化工作完成之后, 进程才会转移到就绪状态。

即使一个进程处于就绪状态, 也只有在操作系统把处理器控制权交给它后才会开始执行, 即图 2-2 中的 2 号状态转换。当前, 进程正在执行, 并处于运行状态 (running state)。如果有两个 (含) 以上的进程处于就绪状态, 则需要从中选择某个进程加以执行, 对应的操作系统部件称为处理器调度程序 (CPU scheduler) 或进程调度器 (process scheduler)。我们将会在第 9 章就进程调度展开详细讨论。

如果一个进程执行到结束或因发生错误或异常而被操作系统中止, 相应的事件 (即一个进程执行到结尾或发生致命错误) 将触发图 2-2 中的 6 号状态转换。这将导致一个进程步入终止状态 (terminated state), 此时此刻, 操作系统将对该进程实施清理操作, 例如, 删除该进程的信息和数据结构, 释放进程所占用的内存和其他资源。当相关清理工作完成时, 就触

发图 2-2 中的 7 号状态转换，从而导致该进程退出系统。

当一个进程处于运行状态时，可能会发生其他两种状态转换，即图 2-2 中的 3 号状态转换和 5 号状态转换。如果该进程所需的某些资源不可用，或者在其可以继续处理之前需要某输入 / 输出操作发生或完成（如等待按键操作或从文件读取数据），就会触发 3 号状态转换。这将导致进程陷入等待状态（wait state）或阻塞状态（blocked state）。对应进程将一直处于等待状态，直到该进程所需资源得到分配或其输入 / 输出请求已经完成。这时，将发生 4 号状态转换，进程从等待状态返回到就绪状态。另一方面，从运行状态直接转移到就绪状态的 5 号状态转换通常发生在操作系统决定暂停正在运行的进程的时候，因为这时系统有更紧要的进程需要运行。这可能是由于定时器，也可能是由于因各种情况而发生的某种其他类型的中断。最常见的理由是因处理器调度算法（我们将在第 8 章进行描述和说明）而要把处理器分配给另一个进程。

27

2.3.2 操作系统维护的进程信息

为了跟踪一个进程，操作系统通常会为其分配一个唯一的进程标识符（process identifier 或 process ID）。同时，操作系统还会创建一个称为进程控制块（Process Control Block, PCB）的数据结构来记录进程的有关信息，包括进程标识符、进程正在使用或请求的资源、进程优先级、进程对各种系统资源或文件的访问权限等。进程控制块还包括对其他操作系统数据结构的引用信息，相关数据结构拥有如何定位内存空间和（进程正在使用的已经打开的）文件的对应信息。对于非运行状态的进程，进程控制块中还保存了该进程的硬件处理器状态的相关信息，譬如程序计数寄存器和其他处理器寄存器的取值，因为在进程重返运行状态和重启进程时将需要这些信息。图 2-3 给出了通常保存在进程控制块中的一些信息。

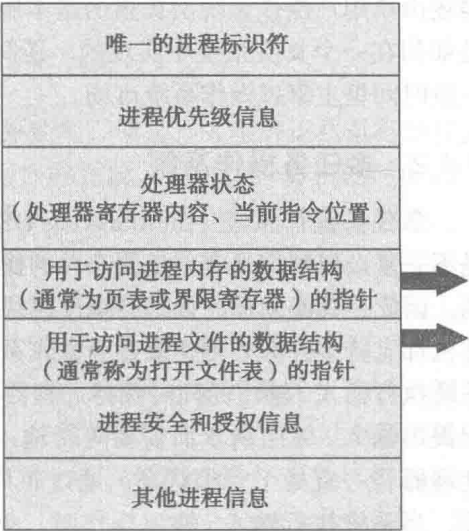
进程正在使用的已打开文件的信息通常保存在一个独立的操作系统数据结构中，该数据结构由操作系统文件管理器模块（见第 12 章）创建和使用。进程占用内存区域的相关信息则通常保存在由操作系统内存管理模块（见第 10 章和第 11 章）创建和使用的页表或界限寄存器中。所有这些表格均被进程控制块数据结构所引用。其他如进程优先级别、进程安全或保护级别（见第 16 章）的引用等信息也将包含在进程控制块中。

2.3.3 进程分类和执行模式

可以把进程划分为如下几种类型：

1) 用户进程或应用进程。它们是那些执行代表用户的应用程序的进程。对应的例子包括运行会计程序的进程、运行数据库事务的进程以及运行计算机游戏的进程。

2) 系统程序进程。这些进程执行的是面向公共系统服务而非特定终端用户服务的应用程序。相关程序一般与操作系统交互密切，并需要关于系统接口的特殊信息以及关于可重定位程序模块或可执行程序文件的布局的系统结构。对应的例子包括程序设计语言编译器和程序开发环境。其他如互联网浏览器、窗口式用户界面以及操作系统



28

图 2-3 进程控制块中由操作系统维护的信息

外壳程序等，在被一些人认为是系统程序的同时，却被其他人认为是操作系统本身的组成部分（见第 2.7 节）。

3) 操作系统进程。它们也被称为守护进程，是执行操作系统服务和功能的进程。对应的例子包括内存管理、进程调度、设备控制、中断处理、文件服务和网络服务。

关于进程，几乎所有的处理器都支持两种执行模式：特权模式和非特权模式（即常规模式或用户模式）。操作系统内核进程通常执行在特权模式（也称为管态、内核模式或监控模式）下，它们被允许执行所有类型的硬件操作，并可访问所有的内存空间和输入/输出设备。其他进程则运行在用户模式（译者注：也称为目态）下，它们常被禁止执行一些命令，如不允许执行低级输入/输出命令等。用户模式还带来了硬件内存保护机制，使一个进程只能访问其受限内存空间中的内存。这样就可以保护操作系统和其他进程所使用的内存，防止相关内存空间因受到错误或恶意的访问而可能导致数据或程序代码的损坏。

2.4 面向功能的操作系统分类

有许多不同类型的操作系统。其中，有些操作系统局限性比较大，仅提供有限的服务和功能；而其他的一些操作系统则非常复杂，提供许多服务和相当多的功能。在这里，我们将就单用户、多任务、分时、分布式及实时系统 5 种操作系统类型进行简明扼要的介绍和说明。

2.4.1 单用户单任务操作系统

单用户单任务操作系统（single-user single-tasking OS）在任何时间仅运行唯一的一道进程。第一批操作系统就属于这种类型，它们是早期的个人计算机上的操作系统，譬如 CP/M 及早期版本的微软 DOS 操作系统（MS-DOS）。类似的操作系统今天依然可能会在诸如嵌入式系统等资源有限的系统中找到。此类操作系统并不像我们下面讨论的其他操作系统那么复杂。但是，它们仍然要处理很多细节和问题。它们提供的主要服务是处理输入/输出及启动和终止程序。鉴于任何特定时间仅有操作系统和一道进程驻留在内存中，所以它们的内存管理相当简单，而且此类系统不需要处理器调度。按照我们的螺旋式方法，我们将在第 3 章中描述由单用户操作系统所提供的基本服务和功能。我们主要以 CP/M 为例具体说明相关概念是如何在一个真实系统中实现的，还间或提到了微软 DOS 操作系统，因为它曾在相当长的一段时间里主宰过操作系统市场。

2.4.2 多任务操作系统

多任务操作系统（multitasking OS）或多道程序设计操作系统（multiprogramming OS）是下一复杂级别即具有一定复杂性的操作系统。此类操作系统对并发执行的多个进程进行控制。因此，其必须设立处理器调度模块用于选择哪一个就绪进程即将运行。大多数现代的计算机都支持多任务。创建多任务操作系统的最初原因之一是为了提高处理器的利用率，即在系统执行输入/输出的同时保持处理器处于忙碌状态。在单任务系统中，如果唯一的运行进程提出输入/输出请求而需要等待输入/输出操作完成，那么在输入/输出操作完成之前，处理器将一直处于空闲状态。通过在内存中同时存放若干道均已准备就绪和可以执行的进程，当系统执行输入/输出操作时，处理器就可以切换到另一进程加以执行。从运行一道进程到转去运行另一道进程称为上下文切换（context switching）。不过，上下文切换开销

很大。完整的处理器状态必须被原原本本地保存起来，以便当进程在后来重新启动时可以正确地恢复。总的来说，当一个正在运行的进程（比如说，进程 A）请求输入/输出且该输入/输出可由输入/输出控制器加以处理时，操作系统的处理器调度模块应检查是否有任何进程处于就绪状态。如果有，则某个就绪进程（比如说，进程 B）将会根据处理器调度算法被选中。于是，操作系统就会保存进程 A 的处理器状态（到 A 的进程控制块中），并（从 B 的进程控制块中）加载进程 B 的处理器状态到相应的处理器寄存器中。然后，在进程 A 转换为等待（或阻塞）状态的同时，处理器将把处理器的控制权交由进程 B，并使其转换到运行状态。其间，进程 A 将始终处于阻塞状态，直到对应的输入/输出操作完成为止。

现在，包括个人计算机在内的大多数计算机的操作系统都支持多任务。即使一台个人计算机常常只有唯一的一个交互式用户，该用户也可以创建多个任务。例如，如果显示屏幕上存在多个窗口，其每个窗口往往是由一个独立的任务或进程来处理的。此外，还有其他任务可能在后台正在运行。一些早期的多任务操作系统只能处理批处理作业，后者是通过读卡器或其他老式的输入/输出设备加载到大容量磁盘上的。许多现在的系统既可以处理批处理作业，也可以处理交互式作业。交互式作业是指那些用于处理用户（通过鼠标、键盘、视频监控显示器和其他交互式输入/输出设备）直接与计算机进行交互的进程。

我们还可以进一步区分出两种类型的多任务操作系统，即通常与单一用户（single user）进行交互的多任务操作系统和支持多用户交互（multiple interactive user）的多任务操作系统。单用户多任务系统包括大多数支持窗口化平台的现代个人计算机。此类系统中，司空见惯的情形是，尽管只有一个用户在与系统进行交互，但该用户可能同时启动了多个任务。例如，用户可能有一个电子邮件程序、一个文本编辑器和一个网络浏览器，且它们在同一时间都被打开，每个程序分别运行在一个独立的窗口中。当前用户聚焦和关注的任务称为前台任务，而其他的任务则称为后台任务。另一类多任务系统同时处理与多个用户之间的交互，故而也被称为分时操作系统。下一小节我们将讨论分时操作系统。

在我们的螺旋式方法部分，我们描述了单用户多任务操作系统的两个例子，一个是第 4 章的手持式掌上电脑（即 Palm Pilot 设备）上的操作系统，另一个是第 5 章的由苹果公司开发的 Mac 操作系统。

30

2.4.3 分时操作系统和服务

多用户操作系统，即分时操作系统，也支持多任务处理，但是，大量的正在运行的任务（进程）是在处理用户与系统之间的交互。之所以称为分时系统，是因为计算机的时间被许多并发交互式用户所“分享”。从操作系统内部机理来讲，交互式进程与批处理进程之间的主要区别是它们对于响应时间的要求。交互式作业通常支持许多短暂的交互操作，并要求系统对每一个交互操作做出迅捷的响应。但是交互式用户的快速响应的需求要求高水平的上下文切换，因而引入了大量的非生产性开销。另一方面，批处理作业不涉及现场的用户，所以并不要求快速响应。因此，仅需要较少的上下文切换，更多的时间则花费在生产性计算方面。分时操作系统同时支持交互式作业和批处理作业，并常常赋予交互式作业以更高的优先级。20 世纪六七十年代，早期的分时系统，如 IBM 的 OS 360 TSO 操作系统[⊖]和 Honey-

⊖ OS 360 TSO 代表 Operating System 360 Time Sharing Option，即带有分时选项的 360 操作系统。

well (霍尼韦尔公司) 的 MULTICS 操作系统, 均支持大量的交互式用户, 不过所有用户都是通过哑监视器和哑终端登录到同一系统的。这是因为终端成本比当时的计算机系统要便宜许多数量级。

伴随硬件和处理器价格的急剧下降, 分时需求逐渐减弱。在现代计算领域, 可以看作交互式分时系统继任者的新一代系统是应用于文件服务器、数据库服务器和万维网服务器 (或称为网站服务器) 中的系统。文件服务器和数据库服务器处理来自于成千上万个用户的访问文件和数据库的请求。相关用户工作在个人计算机或工作站上 (而非坐在与进程运行的服务器相关联的哑终端前面), 并通过网络向服务器发送服务请求。大型的数据库服务器通常被称为事务处理系统 (transaction processing system), 因为它们每秒会处理非常多的用户事务。网站服务器处理关于网络文档的请求, 并经常会从数据库服务器中检索和获取一些文档信息。数据库服务器和网站服务器要求操作系统具备处理成百上千个并发进程的能力。

2.4.4 网络和分布式操作系统

现在的大多数计算机, 或者一直连接在网络上, 或者配备有相关设施使它们可以跟某种类型的网络进行连接和断开。这便允许多台机器之间共享信息和资源, 但同时也要求操作系统为这些网络连接提供相应的功能。相关功能可以划分为两个主要层次:

1) 低级的网络访问服务 (low-level network access service)。操作系统往往具有建立网络连接以及在相互连接的机器之间发送或接收消息等附加功能。

[31] 2) 高级服务 (higher-level service)。用户常常希望能够连接到其他计算机上, 进而浏览信息、下载文档 (文本、图片、歌曲) 或各种类型的程序, 或者访问数据库。这些操作一般通过网络浏览器或特定服务例程 (譬如用于登录到远程机器的 telnet 或用于文件传输的 ftp) 来完成。正如我们先前提到的那样, 这些服务在某些人看来是独立的系统程序, 而在另外一些人看来则属于操作系统的组成部分。

就像我们将在第 15 章所看到的那样, 标准的网络协议实际上会提供若干层级的服务, 从基本的硬件层级到用户交互层级。与网络连接相对独立, 分布式操作系统可以提供一系列各式各样的功能。一个非常基础的分布式操作系统, 有时也被称为网络操作系统 (network OS), 提供了从用户登录的一台所谓客户端 (client, 或称为客户机) 的机器连接到一台所谓服务器 (server) 的远端机器上, 并访问远程服务器的能力。然而, 客户端用户必须知道他们要访问的特定机器的名称或地址。大多数现有的系统至少会提供这样的服务水平。例如, telnet 和 ftp 服务就属于这一层级。

另一种极端的情况是, 一个集大成的分布式操作系统可以支持用户登录到一台客户端机器上, 透明地访问其被授权访问的所有可能的服务和文件, 甚至无须知道相关服务和文件驻留在什么地方。这时, 操作系统本身将负责保存目录信息, 以便定位任何需要的文件或服务等并连接到适当的机器上。这被称为位置透明性 (location transparency)。物理上而言, 相关文件和服务可能在多个系统上进行了复制, 所以操作系统应能够选择最方便访问或最可高效访问的副本, 称为复制透明性[⊖] (replication transparency)。操作系统还应当可以完成动态负载均衡 (dynamic load balancing), 即当选择一台服务器时, 能够选择一台负载不重的机器。

[⊖] 还有分布式操作系统可以满足的许多其他的透明性, 具体参见第 17 章。

这样的操作系统显然是非常复杂的，因此，除了在专用系统或研究原型领域外，尚未真正出现过！

在这两种极端情况之间，人们可以考虑各种类型的分布式操作系统，它们提供的功能比最小功能集大，但比完整的期望功能集小。

2.4.5 实时操作系统

实时操作系统属于多任务操作系统，但其在完成一些或全部任务的截止时间方面有特别的要求。如下为两种类型的截止时间：

1) **硬性的截止时间 (hard deadline)**。一个任务具有一个硬性的截止时间，如 n 毫秒，意味着其必须在提交后的 n 毫秒内完成；否则，错过了截止时间，任务的完成将变得毫无意义，甚至可能导致极其严重的后果。此类任务的例子包括在钢铁厂或炼油厂的工业控制任务，或武器制导系统中的任务。

2) **软性的截止时间 (soft deadline)**。一个任务具有 n 毫秒的软性截止时间，意味着其应当在提交后的 n 毫秒内完成，但是，该截止时间可以错过且不会造成灾难性后果。此类任务的例子，如虚拟现实游戏中伴随用户移动操作而更新显示的任务。

对于硬实时操作系统而言，其调度算法在决定下一个运行的进程时应考虑每个进程的截止时间和估计运行时间。这些操作系统主要应用在飞机或过程控制系统等设备上的嵌入式系统中，其做出关键决策的软件进程必须在指定的截止时间内完成。另一方面，对软实时系统而言，只需要对那些已被指定为实时任务的任务赋予高优先级即可。为此，当前大多数的操作系统，如 Windows 2000 或 Solaris，均提供软实时支持。

[32]

令人遗憾的是，大多数操作系统中已经发展到能够提供顺畅的平均响应时间的大多数技术，无一例外地建立在统计决策的基础之上。这些技术在硬实时系统中将不起任何作用。硬实时系统需要独特的算法来实施时间关键性事件的调度。因此，我们不会花太多的时间来讨论这样的系统。此类系统最好单独介绍。

2.5 操作系统构建方法

2.5.1 整体式单内核操作系统方法

第一批操作系统均被分别写到了唯一的一个程序中。这种构建操作系统的方法称为**内核方法 (kernel approach)** 或**整体式内核方法 (monolithic kernel approach)**，如图 1-3 所示。伴随整体式内核操作系统包含越来越多的功能，其规模也越来越大，在某些情况下，从几千字节增长到了好几百万字节。相对于空间有限且价格不菲的内存来讲，普遍认为操作系统的存储开销（被操作系统占用的内存的百分比）有些太大了。如此臃肿的操作系统不仅占用更多的内存，而且像大多数大型的程序一样，其效率比不过一个较小的系统，同时还有更多的错误且很难维护——无论是添加功能还是修复错误。这使得操作系统设计人员开始基于一个更加模块化和分层设计的方法来开发操作系统。

2.5.2 分层式操作系统方法

发展起来的模块化方法是一种**分层式体系结构 (layered architecture)**。整个操作系统被划分为若干模块，分别限定完成特定的功能，如处理器调度或内存管理。相关模块按抽象递增的次序组织为几层，即每一层提供了一个更为抽象的系统视图，并依赖于它下面的各层的

33

服务。这种分层式方法将会隐藏处理硬件设备的特殊性和细节，并为操作系统的其他部分提供一个通用的抽象视图。因此，当新设备进入市场时，新的设备驱动程序可以被添加到内核而不会极大地影响其他提供内存管理、处理器调度和文件系统接口的操作系统模块。图 2-4 以一种非常初步的方式对此进行了说明。

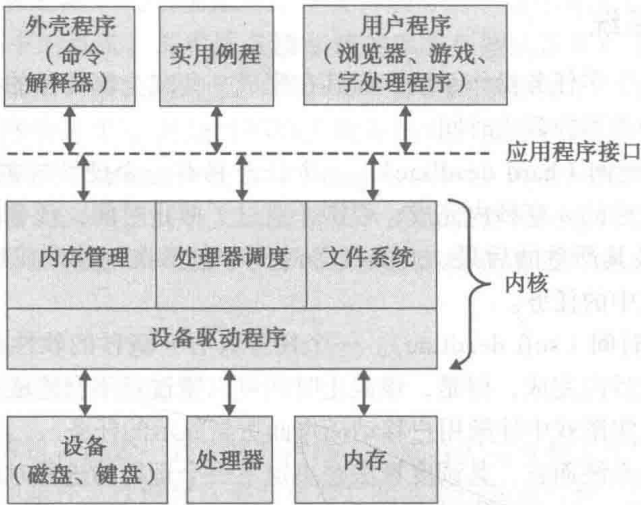


图 2-4 操作系统的分层式模型

这种方法可以扩展到通过若干层来实现一个操作系统。其变种之一是，仅允许 n 层的模块调用下一层即 $n-1$ 层的模块。其另外一个变种是允许 n 层的模块调用任何较低层（ $n-1$ 层、 $n-2$ 层，等等）中的模块。还有一个更进一步的变种是，除允许 n 层的模块调用任何较低层的模块外，还允许 n 层的模块与 n 层的其他模块之间（即同一层的模块之间）进行相互调用。由于要将复杂的操作系统功能划分为多个层次非常困难，所以通常在实际的应用中只使用两层或三层。在后面的章节中，我们将会分析和讨论分层式设计的更具体的实例。大多数现代的操作系统都是基于分层式体系结构建立起来的。然而，一些操作系统编程人员却认为，分层式方法是远远不够的，操作系统的设计应该回归到内核代码的最小化，即应当提倡微内核的理念。

2.5.3 微内核操作系统方法

微内核（microkernel）方法如图 2-5 所示。在微内核体系结构中，包含在微内核中的只有基本功能，通常为各种类型的设备驱动程序的接口。确切地说，在这些模块中仅有的代码是那些必须运行在管态下的代码，因为相关代码实际使用了如保护指令等特权资源，或者访问了不在内核空间中的内存。操作系统功能的其余部分依然是常驻内存操作系统的组成部分，但它们是在用户模式下而非内核模式下运行。在内核模式下运行的代码简直可以做任何事情，所以在这类代码中发生的错误比在用户模式下运行的代码发生的错误会带来更大的危害。因此，微内核理论认为，该方法的好处在一定程度上源自于“运行在管态的代码量越少，则系统越健壮”的事实。同时，微内核方法也使得相关缺陷会更容易被查出。此外，尽管需要为此付出一些额外的设计努力，但却可以使正确的实施变得更为可能。最后，把一个小微内核移植到新的平台上要远比移植一个大的分层的整体式内核容易得多。另一方面，微内核必须利用中断来实施从操作系统的用户模式部分到其特权模式部分的调用。而这些中

断常常需要上下文切换。微内核方法的批评人士说，这会使得微内核操作系统的运行速度慢下来。（应当指出的是，这个议题并没有在操作系统社区得出最终结论。）

34

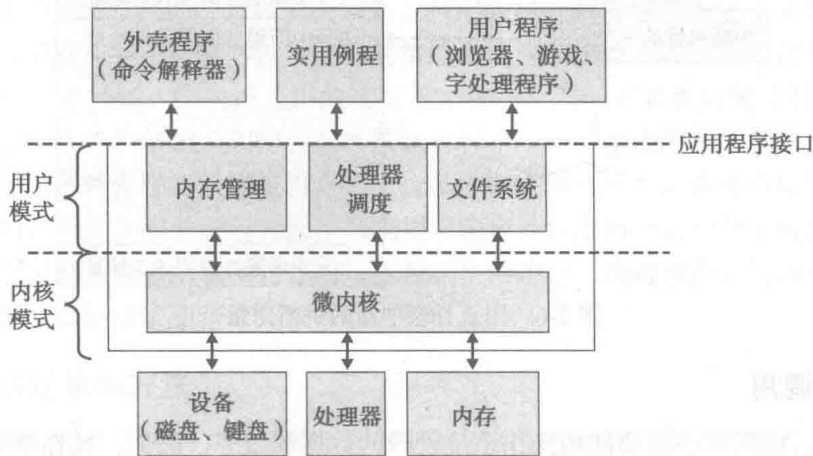


图 2-5 操作系统的微内核模型

2.6 操作系统实现中的一些问题和技术

正如我们在第 2.2 节和第 2.5 节中讨论的那样，操作系统是一个拥有许多模块和部件的复杂的软件系统。与任何此类系统一样，在典型操作系统的内部实现中将会有许多数据结构和算法。在这一节中，我们讨论一些实现技术，它们是大多数操作系统甚至所有操作系统的组成部分。相关主题包括用于处理中断的一般方法、许多操作系统部件中用到的队列和数据结构、面向对象的操作系统实现方法以及虚拟机。

2.6.1 基于中断向量的中断处理

正如我们多次提到的那样，中断（interrupt）是被操作系统利用的一种机制，其用来向系统发送信号，提醒系统发生了某件需要立即关注的高优先级的事件。许多中断事件与输入/输出相关联。一些常见的中断事件，或者以信号告知读或写磁盘块已经完成，或者以信号告知鼠标按钮已被点击，或者以信号告知键盘按键已被按下。正如我们所看到的，大多数此类中断对应于一些硬件操作。硬件通过一个特定的中断号与每个中断事件相关联。当相应的事件发生时，中断控制器通常会将此中断号放置在一个中断寄存器中。根据特定类型的中断事件，操作系统必须采取相应的行动。这里出现的问题是：如何高效地确定发生了哪项特定的中断事件以及如何启动服务于该中断的相应过程？

中断处理的一般技术是采用一种称为中断向量（interrupt vector）的数据结构（如图 2-6 所示）。对于每个中断号，该向量中均有相对应的一个表项。该表项中包含对应中断类型的中断服务例程的内存地址。在中断寄存器中放置的中断号被用作中断向量的一个索引。中断向量表项被硬件提取出来作为地址，同时硬件将以子程序方式来实际调用相应的中断处理例程。当中断例程完成后，其将仅仅是从调用返回，恢复被中断的进程。

35

在只有少许输入/输出设备的小型嵌入式系统中，硬件可能并不提供中断系统，而是选用了一种所谓的状态驱动（status-driven）的系统。在此类系统中，应用程序（或操作系统）基本上就是一个大循环，通过顺次检查每台设备的状态来确认相关设备是否需要服务。

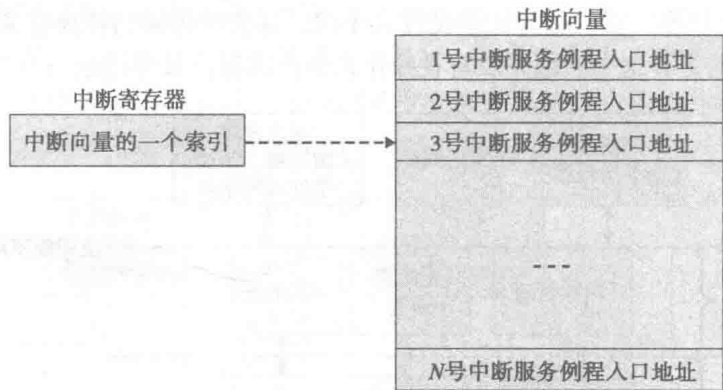


图 2-6 用于处理中断的中断向量

2.6.2 系统调用

一般而言，应用程序需要使用操作系统管理的数据和服务。例如，操作系统通常管理着系统上的所有硬件设备（如声卡），而应用程序并不能直接访问这些硬件设备。同时，应用程序之间可能需要相互通信，于是操作系统不得不承担起中介的角色。

任何普通的应用程序都应当能够使用操作系统管理的数据和服务，且其采用系统调用（system call）的方式来向操作系统请求服务。系统调用非常类似于任何其他的功能调用。首先，应用程序利用描述所需服务的信息来加载某些寄存器，然后执行一个系统调用指令。然而，系统调用指令通常会触发一个即将由操作系统来处理的中断，而不是直接去调用一段用于完成相关功能的代码。操作系统将执行所需的服务，然后将控制返回给应用程序。这种机制还支持操作系统通过首先检查来确认应用程序是否允许以所请求方式访问资源，从而实现某种程度的安全。

一般情况下，应用程序开发系统会提供一个库，该库经过加载后将作为应用程序的组成部分。于是，就由该库负责处理把信息传递给内核以及执行系统调用指令等具体细节。通过该库提供的相关功能，降低了操作系统和应用程序之间的关联强度，提高了应用程序的可移植性。

36

2.6.3 队列和表

操作系统通过管理许多数据结构来完成其各项任务。表和队列是常用的两种数据结构。表是用来存储操作系统管理的各种对象的相关信息的。例如，在第 2.2 节中描述的进程控制块，就是表的一个例子，操作系统通过维护该表来记录和跟踪与每个进程相关联的信息。页表（page table）是另一种常见的表，在硬件支持分页内存管理（见第 11 章）的情况下用于记录进程的地址空间。对于每个进程来说，操作系统会为其维护一个进程控制块和一个页表。还有一种典型的表是打开文件表（open file table），其为系统中打开的每个文件建立和保存了一个表项。

操作系统还维护了一些队列，用于记录按某种方式有序的相关信息。可被多个进程共享的每个资源都需要一个队列来保存该资源的相关服务请求。例如，由于多个进程可能都需要读写磁盘页，所以操作系统就维护了一个磁盘调度队列（disk scheduling queue），用于记录等待磁盘输入/输出的进程列表。打印机服务的请求可以保存在打印机队列（printer queue）中。而准备就绪、等待运行的相关进程的列表则可以保存在就绪进程队列（ready process

queue) 中。

许多这些“队列”并非严格意义上的队列,因为真正的队列总是按照先入先出(First-In-First-Out, FIFO)的原则进行管理的。但是,利用队列的调度算法决定了队列中的元素的顺序。例如,如果选择下一个要运行的进程的策略是优先级策略,那么就绪进程队列的调度程序应当实现相应的策略。在先进先出的情况下,每一个新的元素均应被放置在队列的末尾。当处理器需要执行一个新的进程时,它将从队首移出一个元素来进行处理。正如我们将会看到的那样,存在各种各样的队列组织方式,这具体取决于每种类型队列的特殊要求。

队列中的每个元素都必须包含操作系统用于确定动作取向所需的所有信息。例如,每个就绪队列元素可能包含指向就绪进程的进程控制块的指针。通过该指针访问对应的进程控制块,操作系统就能够检索和获取到所需的进程信息。

2.6.4 面向对象的方法

有一种开发操作系统的方法是基于面向对象软件工程的原理和开发实践,并将其运用于操作系统的设计及实现。根据这种方法,每个操作系统模块应当被设计为一个对象(object)集合,而每个对象应当包含方法(method),用于为操作系统其他部分或应用程序提供服务。采用对象方式来构建操作系统能够提供面向对象软件工程的许多优点,譬如对象数据结构的封装、接口与实现的相对独立、对象重用的可扩展性和易用性,以及许多其他的优势。简单来说,对象的关键特征在于对象的内部结构是隐藏的,而针对对象中所含数据的任何访问都需通过对象的方法来进行。这就使得应用程序不太可能由于误用对象而引发其他模块的问题。

37

已经有多起制作面向对象操作系统的尝试,最著名的要数 NeXT 计算机公司的 NextStep 操作系统(NextStep OS)和 Be 公司的 BeOS 操作系统。一些研究项目也应运而生,最出名的如 Choices、Athene、Syllable、TAJ 以及 JNode(用 Java 语言编写的操作系统)。不过,似乎还未有哪个主要的操作系统是真正基于对象构建的。通常情况下,内核模块采用 C 语言或汇编语言编写,库则提供面向对象接口方式的应用程序接口,从而能够在大多数提供对象支持的高级语言中被调用。Windows NT 就是典型的此类操作系统,其各模块内部的数据结构并不是对象。

2.6.5 虚拟机

另外还有一种关于操作系统设计的方法是利用软件模拟器来抽象或虚拟化(virtualizing)整个系统(包括设备、处理器和内存)的技术。相关概念被称为虚拟机(Virtual Machine, VM)。引入虚拟机的一个主要原因是,其可以保护一个仿真环境避免受到其他仿真环境的影响,因而一个程序的崩溃不会影响和蔓延到其他程序上,即不会导致其他程序发生崩溃。这种被抽象的系统设计可以是一个实际的硬件设计,也可以是一个理想模式的应用程序虚拟机。

硬件级虚拟机

在这种方法中,程序或内核子系统将提供一部实际的硬件机器的软件仿真。其具体包括两种不同类型的仿真,一种是主机硬件系统本身被仿真,另一种则指被模拟的是另外一种处理器。后一种类型往往由制造商开发,通过提供一个程序来模拟旧系统,从而协助客户从一个旧系统迁移到一个新系统。例如,IBM 开发了许多不同类型的仿真软件包,以帮助客户从 1401 系统迁移到普遍使用的 360 系列上。在这种情况下,仿真通常是由运行在用户模式

下的应用程序来完成的。

主机的仿真经常被用来支持多个操作系统内核同时运行，如图 2-7 所示。在这种情况下，仿真是由一个特殊的主机操作系统（host OS）的内核来完成的。该模型支持一个或多个操作系统内核作为客户机操作系统（guest OS）运行在虚拟机层级上。虚拟机层级创建了一个接口来抽象硬件，从而使每个内核都认为自己是在硬件上独自运行的。这些内核可以源自不同的操作系统，也可以是同一种操作系统的不同实例^①。虚拟机模型的主要困难之一是创建一台准确模拟硬件的虚拟机，从而使内核可以像直接运行在真实硬件上一样而运行在虚拟机上（仅仅慢一些而已，因为它们实际上是在与其他内核分享着硬件）。第一批中的一个（如果不是第一个）

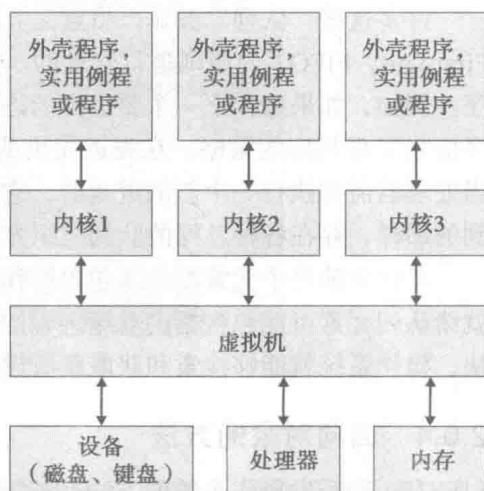


图 2-7 硬件虚拟机

这种类型的仿真软件包是由 IBM 为 360 模型的改进版 40 而创建的，称为 CP-40。其上面运行着客户机操作系统——特别是剑桥监控系统（Cambridge Monitor System, CMS）的多个实例。那个早期的软件包已被重新实现过多次，目前的版本，即 z/VM，运行在 z9 系列主机上。

现在，虚拟机系统已经变得相当普遍了。人们很容易想象，让一个操作系统运行在另一个操作系统上不会特别高效。为此，当代的虚拟机操作系统通常运行的是经过略微修改的客户机操作系统版本，后者很清楚自己是运行在虚拟机环境下并以一种略微不同的方式来处理事情，从而使虚拟机仿真能够更加高效。此外，再新一点的处理器还常常提供了支持虚拟化的额外指令和其他功能。

应用级虚拟机

现在，把术语虚拟机应用到任何创建抽象机的软件的情况随处可见。有时，所模拟的机器并非一个真实的处理器，而是一个试图支持一种特定语言或一大类语言的理想化的机器设计规范。这样的系统有时被称为应用程序虚拟机或应用级虚拟机。一个早期的此类设计被称为 p-code 系统，由加利福尼亚大学圣地亚哥分校设计，用于支持他们的 Pascal 系统。目前非常流行的虚拟机是 Java 虚拟机（Java Virtual Machine, JVM），其创建了一部运行 Java 程序的抽象机。有时，Java 虚拟机以支持 Java 程序执行的独立软件包的方式运行。还有一些情况，虚拟机仿真可能是在另一个程序（如网络浏览器）的内部。这时，Java 程序在可做事情方面有更多的限制。另外一个此类的软件包是由微软开发的通用语言运行库（Common Language Runtime, CLR），用于支持他们的 .net 体系结构。在这种情况下，抽象机被设计用来支持一大类语言（即好多语言），而不是一种单一的语言。

由于虚拟机仿真可能有点低效，所以开发和运行在应用级虚拟机上的代码经常被编译成本地机器码，以便它会运行得更快些。相关技术被称为即时（Just-in-Time, JIT）编译。通过即时编译产生的二进制代码通常在执行后就会被丢弃掉，不过其也可以被永久保存下来供

^① 事实上，IBM 创立虚拟机概念的部分缘由是为了支持操作系统编程人员测试内核，这样，即便正在调试的内核崩溃掉了，其他内核依然可以继续正常运行。

以后重复使用。

2.7 操作系统功能及向后兼容的最小化方法和最大化方法

我们以一场关于“什么功能应当包括在操作系统里？”的讨论来结束这一章。换句话说，操作系统究竟应该做什么呢？这是一个很大的问题。让我们用某种哲学的目光来审视一下这个问题。一个极端是**最小化**（minimalist）哲理，即只有那些真正要进入内核（或微内核）的东西才包含在操作系统里。其他部件则可以添加到库例程或作为“用户”程序（不是内核的组成部分，但通常并不由用户编写）。另一个极端是**最大化**（maximalist）哲理，即把大部分常用的服务都放到操作系统里。例如，如果最大化哲学被采用，诸如屏幕窗口管理等服务就应包含在操作系统内核里，因为几乎每个人都使用该项服务。

最小化主义者争辩说，他们的方法允许每个用户都可以选择他们想要的东西。例如，用户可以从一大群窗口管理器中进行选择，而如果喜欢，事实上可以选择两个甚至更多。这便使得更容易选择部件和建立所要的配置。用户甚至可以编写新的部件。最小化主义者还称，这种方法使操作系统模块易于设计和编码，且更易于调试。他们经常宣称由此形成的系统“更简练”或“更干净”。

最大化主义者反驳道，在一些基本领域用户选择是一个问题，这太灵活了。他们宣扬共同的“外观和感觉”，因为常见的应用程序功能（如滚动条、菜单和移动光标）应考虑到使用上的更一致化和让用户更满意。这可以使用户更容易理解关于应用程序如何工作等基础知识，并建立起应用程序之间的一致性。他们声称，诸如在屏幕上绘画、移动鼠标以及菜单等常见功能，几乎每一个应用程序都会用到，故而应该在一个地方即操作系统里高效一致地完成。他们还断言，有些功能可能在操作系统里更高效，而其他一些功能，譬如安全功能，则必须在内核中完成。

事实上，几乎没有什么操作系统是真的最小化或最大化——和大多数争论一样，选择是由大剂量的“真实世界”注入讨论中做出的。例如，如果我们研究手持小电脑（掌上电脑）的操作系统，许多现实世界的问题就会影响到设计的选择。这些问题包括非常有限的内存，因此应当把尽可能多的功能放到操作系统里，从而通过共享例程使应用程序减少对内存的使用。另一个问题是，制造共同的外观和感觉，但是仅仅包括那些最普遍需要的例程，以避免每个人都需要为不经常使用的服务所占用的内存而支付额外的款项。

40

2.7.1 向后兼容

最后一个问题，向后兼容（backward compatibility，译者注：又称向下兼容，downward compatibility）是成功的代价。这个问题给操作系统的设计人员和实现人员所带来的困难要远远高于能够想象到的情况^①。向后兼容是指在新版本操作系统上运行旧的应用程序的能力。这种能力几乎是任何操作系统的每一个新版本的一个卖点。事实上，甚至没有先前版本的新操作系统也会宣称，能够透明地（无须修改地）运行为其他（流行）操作系统开发的应用程序。请注意，这意味着可执行程序（二进制代码）必须不经改变就能在新系统上运行。

有些系统声称新系统是“源代码兼容”的，也就是说，应用程序从旧系统迁移到新系统，其源码必须重新编译，但不一定发生了变化。这对于购买了程序但却只有可执行文件

① 别忘了，仅仅把系统中的旧代码遗弃就多难！

的人来说没有任何帮助！这不仅要求操作系统的每一个新版本都应当包含以前版本中的所有服务，而且相关服务也必须以相同的方式来工作，即使那些新服务做相同或类似的事情可能更高效和更安全。一个最可怕的问题是，甚至漏洞——那些可能已被发现的——也必须保留下来，因为某些应用程序可能已经利用了那些漏洞的相关功能。例如，在微软的 DOS 系统中有一个闻名遐迩的支持文件大小截断（即让文件大小收缩）的漏洞，就在操作系统的许多版本里保留了几十年（甚至穿越到 Windows 系统）。因为在原先的“有漏洞”的版本里，没有其他方法可以截断文件。不久，该漏洞在一个版本里被修复，但为了兼容已经存在的可执行文件，修复是以扩展（即新服务）方式完成的。旧的服务依旧保留“原有漏洞”。（兼容性问题有时隐藏在下面这句名言里：“这不是一个污点——这是一个特点！”）

2.7.2 用户最优化与硬件最优化

最后一点：个人计算机已经把操作系统的传统目标颠倒过来了。直到个人计算机出现之前，操作系统的主要目标之一始终是优化一堆非常昂贵的硬件的利用率。这意味着应充分利用价格昂贵的内存的每一位（由此曾导致了臭名昭著的千年虫问题，即 Y2K bug）、慢慢腾腾的处理器每个指令周期，以及容量有限但价格不菲的磁盘驱动器的每个扇区。然而，一旦集成电路的水平使个人计算机的生产变得相当便宜，整个系统最昂贵的部分就变成了坐在显示器前面的用户，而不是坐落在显示器后面的装置。这意味着，操作系统需要非常灵敏地响应键盘，并尽可能快速流畅地更新屏幕显示，即使那样做将会伴随处理器的低效使用。图形化用户界面是一个很好的例子。如果我们仍然仅仅使用着每台价值一百万美元的大型机系统，图形化用户界面将是最有可能不再那么普遍的。

[41]

2.8 小结

在本章中，我们首先给出了一个简单的用户场景，并就该场景下操作系统采取的一些动作进行了描述。然后，我们概要介绍了操作系统管理的系统资源的主要类型，并讨论了主要的操作系统模块。接着，我们针对操作系统所完成事项的核心所在，即进程的概念进行了定义，并描述了进程的状态以及由操作系统维护的关于每个进程的一些信息。我们还讨论了不同类型的操作系统的特点，从每次可以执行一个单一进程的系统到管理并发执行进程的系统，再到分时系统和分布式系统。

在此基础上，我们阐明了构建操作系统所采取的一些不同体系结构和方法，具体包括整体式操作系统、微内核体系结构和分层式体系结构。我们讨论了操作系统维护的一些常见的数据结构（如中断向量和队列）、面向对象的系统以及虚拟机。最后，我们以一场关于操作系统功能的最小化方法与最大化方法的哲学讨论而结束本章。

参考文献

Bach, M. J., *The Design of the UNIX Operating System*.
Englewood Cliffs, NJ: Prentice Hall, 1986.
Beck, M. et al., *Linux Kernel Programming*,
3rd ed., Reading, MA: Addison-Wesley, 2002.
Hayes, J. P., *Computer Architecture and Organization*.
New York: McGraw-Hill, 1978.

Lewis, R., and B. Fishman, *Mac OS in a Nutshell*.
Sebastopol, CA: O'Reilly Media, 2000.
Russeinovich, M. E., and D. A. Solomon, *Microsoft
Windows Internals*, 4th ed. Redmond, WA: Microsoft
Press, 2005.

网上资源

<http://developer.apple.com/technotes/>
<http://www-03.ibm.com/systems/z/os/zos/index.html>
(IBM大型机操作系统)
<http://www.academicresourcecenter.net/curriculum/pfv.aspx?ID=7387> (Microsoft® Windows® Internals, Fourth Edition: Microsoft Windows Server™ 2003,

Windows XP, and Windows 2000 by Russinovich, M.E., and D.A. Solomon)
<http://www.linux.org> (Linux内核开发之家)
<http://www.kernel.org> (史上重要的内核源码库)
<http://www.osdata.com> (操作系统技术比较)
<http://www.tldp.org> (Linux文档化项目)

习题

- 2.1 哪些是操作系统必须管理的资源类型?
- 2.2 程序与进程之间的区别是什么?
- 2.3 进程可能的状态有哪些?
- 2.4 多少个进程可以同时处于运行状态?
- 2.5 什么类型的事件可以触发从运行状态到终止状态的转换?
- 2.6 至少给出进程可能等待的几件事情的名称。
- 2.7 进程控制块中存放的一些信息并不总是及时更新。请举出一些此类信息的例子。
- 2.8 列举用户进程、系统进程、操作系统进程三类进程的一些例子。
- 2.9 本章讨论了操作系统总体设计的5种不同类型。如下实例分别属于什么设计类型?
 - a. 手持式计算机和掌上电脑上的操作系统
 - b. UNIX
 - c. Novell Netware
 - d. 磁带录像机 (Video Cassette Recorder, VCR)
 - e. 汽车发动机
- 2.10 如果我们编写应用程序, 需要操作系统为我们管理硬件的理由有哪些?
- 2.11 我们把操作系统划分为独立模块的理由有哪些?
- 2.12 什么是“微内核”操作系统?
- 2.13 一般而言, 面向对象的程序设计的效率比过程化程序设计要低些。为什么我们想用低效率的工具来制造操作系统?
- 2.14 当操作系统从设备接收到中断时, 其通常采用什么机制来选择用于处理中断的代码?
- 2.15 应用程序如何请求操作系统做一些事情?
- 2.16 大多数现代的操作系统都是虚拟机操作系统, 目前这种状态是由操作系统的演化造成的。这是真的还是假的?
- 2.17 现代的软件虚拟机体系结构有哪两种?
- 2.18 你认为操作系统应当包含许多公共的系统功能, 还是应当仅包含最少量的功能而把尽可能多的功能放到另外的层级和库中? 请对此做出解释。
- 2.19 什么是应用程序设计可以依赖的最标准的操作系统应用程序接口?

42

43
44

渐进式构建操作系统： 面向广度的螺旋式方法

正是本书的第二部分才使得这本书与众不同。其他书籍往往倾向于就围绕典型操作系统不同方面的一系列分离的主题按纵深方向、彼此孤立地分别展开讨论。本书则不然，在这一部分将呈现一些章节，针对所选定的操作系统，阐述它们是如何伴随底层硬件的发展及用户期望与需求的增长而被迫演化的。这些入选的系统都运行在某种类型的个人计算机上。之所以这样做并非无意之举，而是经过深思熟虑的。这在一定程度上是考虑到大多数学生熟悉相关的计算机，并可能在之前已经见到过许多这样的机器和操作系统。这样的系统也是学生最有可能接触到的系统，至少是更现代的系统。同时，个人计算机操作系统的演化与更大些的机器上的操作系统的演化是并行展开的。为此，个人计算机操作系统的实例，从最原始的到最复杂的都普遍存在。而许多这样的操作系统也存在于更大些的机器上，包括现在一些最大的大型机上也能找到。

第二部分包括 5 章。第 3 章讨论了一种早期的个人计算机操作系统，即 CP/M。这是一种单用户、单任务操作系统，且没有图形化用户界面（Graphical User Interface, GUI）。其仅支持平坦型文件系统（即单级文件系统）。就这点来说，其非常类似于早期的大型机操作系统，如 IBM 709x 系列机上的 IBSYS。我们阐明了在这些简单系统中的操作系统应有的所有基本机制，包括内核与操作系统的分离以及对文件系统的支持。

在第 4 章，我们研究了一个引入另外两种理念的操作系统：一是同时运行多个程序的想法，二是图形化用户界面的使用。所涵盖的操作系统为许多掌上电脑和手机上都在使用的 Palm 操作系统（Palm OS）。这两项额外的要求必需有额外

的操作系统机制加以支持，尤其是处理器抽象和进程控制块。掌上电脑和手机系统通常都没有辅助存储设备，但它们仍然支持文件系统的概念，因为应用程序员对“隐喻修辞方法”（译者注：此处应当指把内存当作外存使用来支持文件系统的做法）非常熟悉。它们确实有一个图形化用户界面，但屏幕的使用受到对应非常小的屏幕尺寸的限制。我们还讨论了这两种限制情况对操作系统设计的影响。

第5章讨论的操作系统系列则又引入了其他一些需求。这就是麦金塔（Macintosh，或简称为 Mac 或 MAC）系列操作系统，其设计从一开始就关注到了辅存。该族操作系统的进化很有意思，因为其本身就是螺旋式演化的一个例子。Mac 操作系统最初就已提供但在 Palm 操作系统部分未予讨论的唯一特征是，Mac 操作系统的图形化用户界面支持重叠窗口。其仍然是单用户系统和平坦型文件系统，就像 CP/M 和 Palm 操作系统一样。然而，伴随 Mac 操作系统的升级演化，苹果公司（Apple）又为其增加了许多新的功能，如多任务、多级文件系统（Hierarchical File System, HFS，或称为分层式文件系统）、多用户（尽管不是同时/并发支持）、多处理器以及虚拟存储系统。这些机制将逐一依次展开讨论，且由最后的虚拟内存主题自然而然地引出下一章的内容。

第6章涵盖了 Linux，以作为适应从嵌入式系统到实时系统再到超级计算机等许多不同硬件平台的操作系统的实例。在这里，Linux 与前述系统的主要区别是，它是基于在多个终端上的多个用户的假设前提而设计的。为了支持相关功能，操作系统内部（特别是在文件系统中）必须提供更多的保护机制。Linux 也是开源操作系统的一个例子，这项不同在本章中也会有所探讨。在本书后面的部分将会就 Linux 展开进一步更详细的介绍。

第7章就跨越多个计算机系统的操作系统的设计所引发的问题进行了探讨。通常，此类系统会跨越行政管理领域。几乎可以肯定地说，所涉机构的政策和利益往往并不一样。事实上，彼此间甚至可能相互冲突。尽管如此，有关机构还是找到了某些共同的利益，从而驱使他们建立起了跨越此类界限的系统。本章将通过 GLOBUS 来具体阐述有关的问题。当然，也会讨论其他的一些系统。

简单的单进程操作系统

现在，我们开始本书的“螺旋式”部分，每一章将基于一个特定的真实的操作系统来讨论一种类型的操作系统。我们从一种真实但简单的功能有限的操作系统出发，并会在后续的章节中渐进式地讨论更为复杂的操作系统。我们将立足于一种早期的个人计算机操作系统（即 CP/M）的功能及经常用于运行该系统的硬件来陈述本章的大部分内容。我们讨论了这些操作系统是如何设计的，以及设计背后的基本原理。虽然这些系统是仅支持单进程且功能有限的系统，但它们为在数以百万台的个人计算机上编写的成百上千个应用程序提供了充足的动力。因而，它们提供了简单操作系统的一个很好的实例。在这一章中所讨论的问题，如输入/输出管理、文件系统、内存和进程管理，将在后续的章节中伴随更复杂操作系统的引入逐渐展开进一步的详细介绍。尽管如此，我们在这里先夯实一个基础：真实但简单的功能。

本章组织如下：第 3.1 节介绍了简单操作系统的前身，即所谓的监控程序，并讨论了它们是如何因应标准化的需要而演变成早期的操作系统的。在第 3.2 节中，我们描述了这种类型的操作系统所使用的早期个人计算机系统的特征。然后，我们在第 3.3 节中，讨论了输入/输出在这样一个早期的操作系统里是如何进行管理的。接下来，我们分别在第 3.4 节和第 3.5 节中给出了文件系统及进程和内存管理的描述。

当时的系统局限性很大，它们一次只运行一个用户程序。进程管理最初仅限于加载和启动一个特定的应用程序。后来的新版 CP/M 增加了后台打印功能。这套机制是多处理概念的开端，后者在现代的操作系统中得到了贯彻实施。操作系统中的内存管理仅限于确认和处理哪一部分的内存用于存放操作系统、中断向量、用户程序或是程序数据，等等。但是，由于内存非常有限，大型程序往往无法完全加载到内存中，或者在它们可以处理的数据量方面存在着极大的局限性。为此，应用程序员就不得不把这样的程序分解成若干部分，并根据需要用其另一部分来替换内存中的部分。这项内存管理技术，称为覆盖（overlay），将在第 3.5 节中讨论。此外，这些技术还预示着在现代操作系统中能够找到的更复杂的内存管理技术。

47

3.1 监控程序和 CP/M

我们从关于“为什么出现了构建个人计算机操作系统的需要？”的讨论出发，来开启本节的内容。这些操作系统的前身被称为**监控程序**[⊖]，其功能非常有限。监控程序没有任何可以遵循的标准——早期个人计算机系统的制造商经常各自编写自己的监控程序，且采用的是独自特有的命令和编程惯例。这种多样性意味着，早期的应用程序必须得为每个监控程序重写代码。

3.1.1 监控程序：简单操作系统的前身

当个人计算朝气蓬勃向前发展且单芯片微处理器使小型且相对便宜的计算机的构建成为可能的时候，软件危机发生了。廉价的微处理器的出现催生了小型初创公司，这些公司把配

⊖ 我们正谈论的是作为操作系统前身的一种软件模块，而不是视频显示终端，后者有时也称作“monitor”（即监视器）且经常用在计算机上。

套元件卖给了本地的那些想要搭建自己计算机的爱好者。相关组件通常包含一个电路板、一个微处理器、一些内存以及一些扩充的设备控制器芯片。这些扩充芯片用于控制各种输入和输出设备，譬如盒式磁带、软盘、外接的视频终端以及打印机。许多公司都在销售个人计算机元器件。不过，无论以何种标准来说，一开始他们都具有极大的局限性。在早期的这类系统中，内存大小也就 1KB~4KB，有时甚至更少。应用程序是用机器语言或汇编语言编写的。一般情况下并没有操作系统。不过，倒是有一个不大的**监控程序**（monitor program）通常存放在只读存储器（Read-Only Memory, ROM）中，可以支持应用程序完成一些简单的、共性的任务，例如：

- 输出字符到视频显示器或电传打字机等设备上。
- 从键盘设备获取字符。
- 把内存的全部或部分内容保存到盒式磁带或软盘上。
- 用磁带或磁盘上存放的镜像对内存内容进行恢复。
- 打印字符到打印机上。

48

监控程序只能完成这些基本的任务，除此之外，没有什么其他的功能。

应用程序可以通过以下步骤调用监控程序，从而把字符（如一个“1”）显示到控制台（console）的视频显示器或电传打字机上。

1）把字符放入监控程序指定的特定寄存器中（假设为寄存器 E）。在本例中，十六进制数值“31”（即字符“1”的 ASCII 码）被存入寄存器 E 中。

2）选择特定的监控程序函数（本例中，“显示字符”的函数对应数值为“2”），并把所选监控程序函数对应的数存入寄存器 C 中。

3）最后，通过 5 号中断执行对监控程序的调用。这将会使监控程序按照存放在寄存器 C 的函数编码和存放在寄存器 E 的字符来执行所调用的显示函数。

4）监控程序输出该字符后，会返回一个表示正确（OK）或不正确（not OK）的状态码放在寄存器 A 中。不正确的话，还会说明发生了什么异常情况，如设备不响应或传入函数的某些参数值非法等。按理说，应用程序须检查寄存器 A 中的状态码，以确定因应特定错误的适当动作。不过，典型的早期应用程序并不总是进行错误检查，因为对于大多数的错误来讲，它们几乎没有什么可做的事情。

3.1.2 为什么创建 CP/M？什么是软件危机

许多公司都在组装和搭建计算机，且每家公司均须提供用作小型监控程序的一套软件。从内存需求角度而言，这些监控程序并不算大，也就几百或几千字节。通常情况下，一个监控程序仅提供十几项功能，但这些功能需要时间和专业技能来开发、调试和构建。更为糟糕的是，没有一个标准的监控程序或监控程序接口。每家制造厂商都只是简单地实现了他们想象中的程序员们想要的功能。例如，在一个监控程序中可能采用寄存器把参数传递到函数中，而在另一个监控程序中则可能采用内存单元来传递参数，但在第三个监控程序中还可能混合使用上述两种方法。这便给应用程序设计编写带来了一个问题。如何才能编写出具有**可移植性**（portable）的程序呢？也就是说，如何才能让写出的程序可以运行在不同制造商的计算机上呢^①？由于监控程序的不同，所以对于每一家制造商的计算机，即便是同样的应用程

① 因为早期的应用程序都是用机器语言编写的，所以这里的应用程序可移植性要求计算机所用处理器必须相同或至少兼容，也就是说，一种处理器的指令集必须是另一种处理器的指令集的超集。

序也需要分别专门编写。这种状况诱发了 CP/M (Control Program/Monitor, 控制程序 / 监控者) 的研发。该系统是基于英特尔 8080/8085 (Intel 8080/8085) 处理器电路的微型计算机创建的, 且 CP/M 最初是数字研究公司的 Gary Kildall 一个人的成果。

3.1.3 CP/M 的构成

CP/M 系统允许软件开发人员、用户和制造商拥有一个唯一简单的标准接口。其通过利用基本输入 / 输出系统 (Basic Input/Output System, BIOS) 软件层, 把硬件设备同操作系统进行了隔离。该 BIOS 和监控程序有些类似, 但具有标准化的功能和接口。每家制造商可以调整 BIOS 来支持和适应他们的特定机器所包含的所有设备^①。但是, 不管底层的设备是如何工作的, BIOS 的接口却是相同的。这样, 要把 CP/M 移植到一个新的系统上, 主要的工作就是根据硬件重写 BIOS 的相关例程。

49

操作系统核心称作基本磁盘操作系统 (Basic Disk Operating System, BDOS), 这就是我们今天所说的内核。其与硬件无关, 并调用了 BIOS 中的更原始的服务。对于 CP/M 运行的任何系统平台来说, BDOS 都是一样的。这种提供了通用系统函数和隐藏了繁琐硬件细节的标准化接口称作抽象 (abstraction)。在本书中, 我们会多次提到抽象技术。

操作系统的最后一部分是称作控制台命令处理程序 (Console Command Processor, CCP) 的操作系统用户接口。CCP 执行的其他命令大部分是磁盘上的程序。CP/M 操作系统的这三部分都很小。每部分的大小也就 2000~4000 字节, 整个 CP/M 都存放在软盘的几个扇区上以便启动计算机。

CP/M 实际标准的存在激励着软件编写人员为各种各样的制造商构建的个人计算机开发应用软件。相关软件能够支持许许多多的输入 / 输出设备, 譬如不同容量的软盘、硬盘以及视频终端。应用程序无须再为每种类型的计算机定制式专门编写。很短的时间内就可以编写完成数百个程序。对于程序员来说, 有文本编辑器、各种编程语言的编译器以及调试工具。字处理程序、会计软件包、简单文件系统、游戏以及许多其他程序的编写还催生了蓬勃发展的个人计算机市场。由于操作系统设计得很好, 在每一层都有一个清晰标定的接口, 所以还出现了控制台命令处理程序 CCP 的若干替代程序可以方便提供不同的接口。

当 IBM 决心进入个人计算机市场, 最初的决定是采用行之有效的 CP/M 标准。由于 IBM 个人计算机 (Intel 8088) 的处理器与 CP/M-80 并不完全兼容——它们分别基于 Intel 8080 处理器和 Zilog (译者注: 美国齐格洛公司, 著名的 Z80 系列处理器的设计者) Z-80 处理器——所以, 进行了一些小的改动。IBM 的硬件非常出名且很有特点, 所以 BIOS 可以充分利用相关特征^②。

在随后的几节, 我们对早期的 IBM 个人计算机和 CP/M 计算机的硬件进行了大胆地概括。我们的目的不是讲授 CP/M, 而是以它作为一个例子来说明简单操作系统的特征和功能。

3.2 简单的个人计算机系统的特征

早期的个人计算机系统包括一块主电路板, 即个人计算机的主板 (motherboard)。这块

① 也可能是由电脑爱好者自己来完成相关适应性调整的, 因为伴随有关软件还有指令集和示例。而且, 甚至像增加系统内存等比较简单的一些事情也要求重新创建基本输入 / 输出系统。

② 最后, IBM 采纳了微软开发的 MS/DOS 操作系统用在他们的个人计算机上。

50

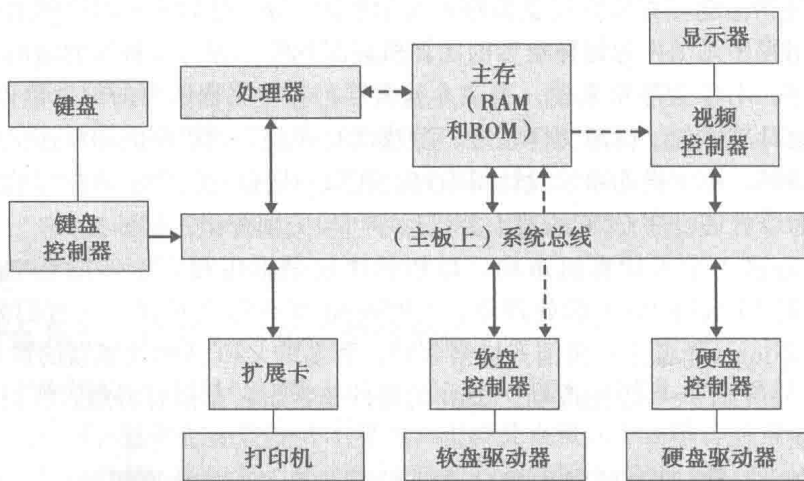
主板有一个微处理器芯片（CPU）、一些随机访问存储器（Random Access Memory, RAM, 或称为随机存取存储器）、包含基本输入/输出系统（BIOS）的只读存储器（ROM）以及把这些芯片连接到一起的若干其他集成电路（Integrated Circuit, IC）。主板还有一些插槽，用于插入外加的扩展电路板（在早期的个人计算机术语中称为插卡，即 card）。这些扩展卡中包括一个视频控制器，通过将显示器电缆插入视频控制卡中就可以连接到一台视频监视器上。其他扩展卡包括追加的内存条以及软盘控制器和硬盘控制器。用户输入/输出经由视频监视器和键盘来完成。键盘是直接插到主板上的，主板有一个内置的键盘控制器芯片。另外，主板上还有一个简单的时钟或定时器芯片。

关于早期个人计算机系统典型硬件组成的简单系统示意图如图 3-1 所示。就这类系统而言，其对操作系统设计有重大影响的一些特征如下：

1) 主存大小相当有限。这导致了一次仅加载一道应用程序进入内存的操作系统设计决策。因为 CP/M 操作系统非常小，所以其常驻内存，具体包括装入程序、中断处理程序和设备驱动程序。如果剩余的可用内存无法容纳和放下一道应用程序，则该应用程序将不得不被划分和写成若干可以分别单独放入内存的节。当需要新的一节时，应用程序可基于所谓覆盖（overlay）的内存管理技术用新的一节把旧的一节替换掉。

2) 磁盘格式是标准化的。早期个人计算机的软盘和硬盘的盘块大小及格式是固定不变的。这导致了基于标准磁盘格式的标准化文件系统的设计。

3) 中断处理主要针对输入/输出设备。既然每次只能运行一道应用程序，所以没有必要在应用程序间进行切换。操作系统中无须处理器调度。主要的中断类型均用于处理输入/输出设备。



51

图 3-1 早期个人计算机系统的硬件组成

3.3 输入/输出管理

相对于如今的各种各样的设备来说，当时的输入/输出设备类型很少，所以早期操作系统中的输入/输出处理功能非常有限。早期个人计算机的大多数应用程序一般需要如下输入/输出服务：

- 1) 从键盘读取字符。
- 2) 写出字符到显示器屏幕上。

3) 打印字符到打印机上。

4) 利用磁盘文件系统创建一个新文件、读/写文件以及关闭文件。

许多程序面临的一个问题是,在处理键盘输入和屏幕输出方面缺乏灵活性。因为有许多不同的公司都在制造工作方式各不相同的计算机硬件,所以操作系统试图提供一种标准的方式来处理这些差异。

另外一个问题是性能。具体而言,直接调用 BIOS 或硬件来执行输入/输出指令往往比调用适当的操作系统命令要更快和更灵活。于是,便产生了可移植性(portability)和灵活性及更快速度的折中问题。如果应用程序设计人员仅仅使用操作系统调用来执行输入/输出,则会更好地满足可移植性。而如果应用程序设计人员直接调用 BIOS 和硬件功能,则可能获得更好的灵活性及更快的速度。作为示例,我们围绕早期系统中两个最常见的输入/输出设备,即用作输入设备的键盘和用作输出设备的视频监视器,讨论了有关折中问题。

3.3.1 键盘输入——可移植性与灵活性

有许多类型的键盘。它们可能有 65~95 个键,且在键盘上摆放的位置也不尽相同。从键盘传输的数据可能是串行的,也可能是并行的,而字符则可能表示为 7 位或者 8 位。这如何才能标准化呢? BIOS 是针对每种类型的键盘而分别定制的,但会为操作系统的其余部分提供相同的 BIOS 接口函数集合。于是,BDOS 可使用 BIOS 的那些函数来为键盘创建一个简单的操作系统接口。关于键盘的这些函数(即操作系统的系统调用)是:1)从键盘上读取一个字符;2)检查是否有一个按键被按下。对于许多应用程序来说,这已经足够了。同时,如果一个应用程序就其键盘输入使用了这些标准函数,其便可以移植到任何计算机系统上。

但是,有些应用程序需要额外的灵活性。例如,字处理程序可能想要使用“修正后的”按键,具体而言,用“Control+S”来保存文件,用“Control+C”来弹出一个命令列表菜单。这些特殊的按键或按键组合产生了一个问题,因为它们没有受到 BDOS 的认可,因而无法传递给应用程序。更为糟糕的是,一些按键组合,如“Control+C”可能已被 BIOS 或 BDOS 进行了解读,并可能导致诸如操作系统重启之类的动作。在这种情况下,显然无法把按键组合传递给应用程序。

52

想要额外灵活性来处理键盘的应用程序(因而按键组合对该应用程序来说是有特定意义的)绕开了 BDOS。这一点微不足道。这可能意味着将从 BIOS 而非 BDOS 读取按键,甚至可能从键盘硬件(实际上是键盘接口芯片)直接读取按键。因为在早期的系统里没有内存保护,所以绕过操作系统(BDOS)非常容易。任何应用程序都可以寻址内存的任何部分。使用 BIOS 调用就像使用 BDOS 调用一样简单,只不过基本磁盘操作系统 BDOS 调用不会按应用程序所需的方式来行事而已。这种方法的问题是,程序将不再是可移植的,尤其是在应用程序直接调用到硬件的情况下。

3.3.2 视频监视器输出——可移植性及功能与性能

屏幕或视频监视器带来了更为严重的问题。首先,通过 BDOS 及 BIOS 的接口函数提供的功能非常有限。视频系统的许多功能无法直接通过简单的操作系统调用来使用。例如,人们不能使用颜色、写入多个视频内存“页面”进而通过快速显示一系列图像来模拟运动,也不能把光标移动同写屏幕独立开来。其次,甚至更为严重的是,使用 BDOS 的屏幕输出非

常缓慢。许多应用程序更倾向于把字符直接写到屏幕内存和直接访问视频控制器硬件。也有许多应用程序使用 BIOS 调用来移动光标。它们绕过 BDOS 的主要原因是为了提高应用程序的性能。

直接写入视频内存不仅提供了更多的功能，而且也比通过操作系统系统调用要快了许多。根据所采用的编程语言，它能够快出 100 倍甚至更多！绕过操作系统来显示字符造成了与绕过键盘所导致的相同类型的可移植性问题。然而，性能方面的优势是如此显著，所以使许多应用程序忽略了可移植性来提高性能。这对于游戏类程序而言尤其如此，它们总是试图从硬件里榨取任何可能的哪怕是一丁点的性能。游戏总是推动着个人计算机硬件的迅猛发展和不断进步。

例如，把一个白色的字符“+”写到黑色的屏幕背景中仅需要调用一条单一的机器指令：“Move 0F800, 2B07”。直接把文本写到视频内存是相对直接和简单的。视频内存从十六进制地址 0F800 开始，其对应屏幕左上角的第一个可见字符。紧随地址后面的是对应字符的视频属性（video attribute）。对于彩色适配器而言，其是一个 8 个二进制位的信息：3 位用于表示前景颜色信息^①，3 位用于表示背景颜色信息，1 位用于指示是否为“高强度前景”（即高亮），剩下的 1 位用于指示字符是否闪烁。为此，字符“+”（对应 ASCII 码为“2B”）写到屏幕左上角位置（对应视频内存地址 F800），同时设置前景为白色（“7”）、背景为黑色（“0”）。在白色背景上的一个黑色字符，其视频属性可简单地设置为“70”。

53

3.4 磁盘管理和文件系统

既然有如此之多的应用程序绕过了关于键盘和视频的操作系统系统调用，那么操作系统真正提供了哪些功能呢？此类早期操作系统提供的一项主要服务是标准的可移植的文件系统，即实现磁盘管理的基本功能。大约 75% 的操作系统系统调用是与磁盘文件相关的。在这一节，我们讨论 CP/M 的文件系统，但将首先阐述作为文件系统基础的磁盘系统。

3.4.1 磁盘系统

早期的个人计算机系统，有一套硬件磁盘设备（即 8 英寸软盘）的标准与文件系统一起被使用，如图 3-2 所示。该软盘在中间有一个孔，软盘驱动器通过这个孔把软盘放置在一个主轴上。主轴连接到一个磁盘驱动器马达上，后者将按每分钟 360 转的速度转动磁盘。标准磁盘拥有 77 条磁道（track），编号从距离中心孔最远外圈的 0 号磁道一直到最里面的 76 号磁道。磁道是同心圆，也就是说每条磁道从开始一直到结束与中心孔之间都是等距离的。一条磁道包含 26 个扇区（sector，有时称为块，即 block），第一个扇区编号为 1，最后一个扇区编号 26^②。每个扇区包含 128 字节的数据，另加一些控制信息，譬如扇区编号。软盘是双面的，不足为奇，两面的编号分别为 0 和 1。在中心孔附近，有一个称为指示孔（index hole）的小孔，可被磁盘控制器用来确定各磁道的第一个扇区。所有的磁道都有 26 个扇区，外圈的磁道较长而内圈的磁道较短，但每条磁道均存放有相同数量的数据^③。

① 颜色采用每位分别指示红、绿、蓝的方式来表示；于是白色表示为所有 3 位均为 1（译者注：即二进制数“111”，对应十进制数为 7）的情形。

② 注意：磁道编号从 0 开始，扇区编号从 1 开始。

③ 这对于现代硬盘而言不再成立了。

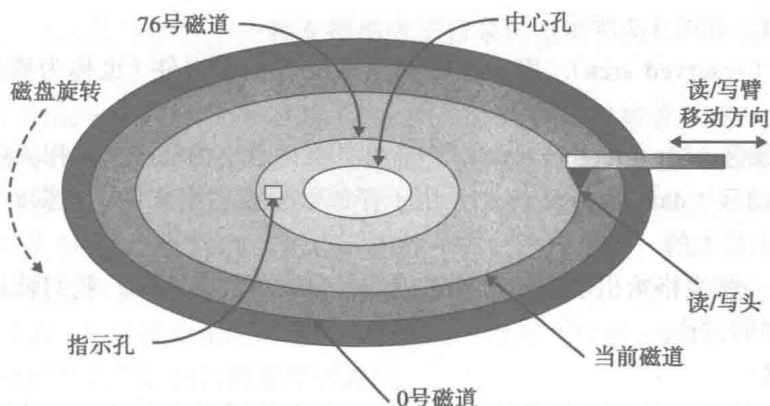


图 3-2 早期个人计算机的软盘

54

磁盘系统是由**磁盘驱动器**（disk drive）和**磁盘控制器**（disk controller）组成的。磁盘驱动器固定和旋转真正包含所存储数据的磁盘介质（软盘）。磁盘控制器则往往建造在计算机主板上或一块插入主板的电路卡上。磁盘驱动器可以在磁道之间移动**磁头**（disk head），磁头中包含有读/写的磁传感器。要在磁道上读取任何单个扇区，磁盘驱动器必须等待该扇区旋转到磁头下。

磁盘控制器可以接受在磁盘的指定面和指定轨道上读取或写入一个扇区或多个扇区的命令。磁盘驱动器的磁头移动马达移动磁头，有时可能会错过指定的磁道（而到了错误的磁道上）。磁盘控制器会检查到这类事件（每条磁道的每个扇区上都有对应的磁道号），并正确地重新定位磁头。有时扇区也可能被错误地读取，同样，磁盘控制器会检查出此类事件，并尝试再次读取。磁盘控制器在指示孔旋转错误后，也会启动寻找磁道开始扇区以重新定位自身。所有这些活动对于操作系统甚至 BIOS 都是不可见的。它们或者在磁盘控制器本身的硬件级别上实现，或者在控制“磁盘控制器上的专用处理器”的软件中实现。这种嵌入在控制器上 ROM 中的软件通常被称为**固件**（firmware）。

早期个人计算机系统的此类标准磁盘，在用操作系统格式化之后，可以存放： $77 \text{ (磁道/面)} \times 26 \text{ (扇区/磁道)} \times 128 \text{ (字节/扇区)} \times 2 \text{ (面)} = 512\,512$ 字节的原始数据（约为 500KB）。**磁盘格式化**（disk formatting）就是把控制信息写到磁盘和把磁道划分为扇区的过程。这种标准盘被用于作为实现个人计算机的操作系统之文件系统的基础，下一小节我们将讨论文件系统。

3.4.2 文件系统

这个操作系统有一个建立在 BIOS 之上的简单的文件系统，用于存放用户文件和系统文件。可存放在磁盘上的系统文件部分包含二进制的操作系统代码本身。此外，每个磁盘设立有一个目录存放了存储在该磁盘上的所有文件的相关信息，包括它们的大小、物理磁盘上存储位置（扇区），等等。存储在磁盘上的文件可能包含如下类型的数据：

- 应用程序生成的数据（文档、数据表、源代码）。
- 应用程序可执行文件（二进制代码）。
- 目录信息（文件名、创建日期、在磁盘上的存储位置）。
- 操作系统的二进制可执行文件（操作系统可执行文件，用于加载或“引导”操作系统）。

为了适应和容纳存储在磁盘上的各种不同类型的信息，就 BIOS 而言，每个物理磁盘被

划分为三个区域，如图 3-3 所示：

- 保留区（reserved area），用于存储操作系统可执行文件（也称为磁盘引导区（disk boot area））。
- 文件目录区（file directory area），包含存储在磁盘上的每个文件相关信息的目录项。
- 数据存储区（data storage area），用于存放数据或程序文件，占据磁盘的其余部分，是磁盘上最大的一部分。

BIOS 内嵌一张表格给出了上述这些区域的各自大小。接下来，我们就这些区域的内容逐个展开更详细的讨论。

磁盘引导区

文件系统的最简单的部分就是这个保留区，其存放了用于启动个人计算机的操作系统二进制代码。这个区域对于文件系统是不可见的，没有对应的目录项，也没有名称。BIOS、BDOS 以及 CCP 的可加载映像都写在这一区域，从 0 号磁道的 1 号扇区开始，扇区挨着扇区、磁道挨着磁道地连续存放。它们均非任何“文件”的组成部分，而只是占用着磁盘的最开始的几条磁道。一般情况下，BIOS 为 2KB，BDOS 为 3.5KB，而 CCP 为 2KB，所以操作系统的二进制代码共同占用了前三条磁道。

当计算机打开或重启时，只读存储器中的一小段代码就会运行，把操作系统可执行映像从磁盘复制到内存，然后开始执行相应程序，即操作系统。这个过程称作引导（booting）或操作系统加载（OS loading）[Ⓐ]。

文件目录区

目录区的大小是固定不变的，被记录在 BIOS 的表格中。对于软盘来说，目录最多可包含 64 个目录项，每个目录项为 32 字节[Ⓑ]。磁盘目录项设计如图 3-4 所示。目录中的每个目录项均包含如下信息：

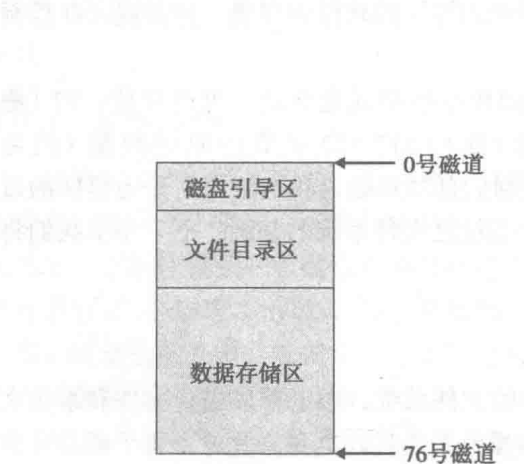


图 3-3 典型的 CP/M 文件系统布局

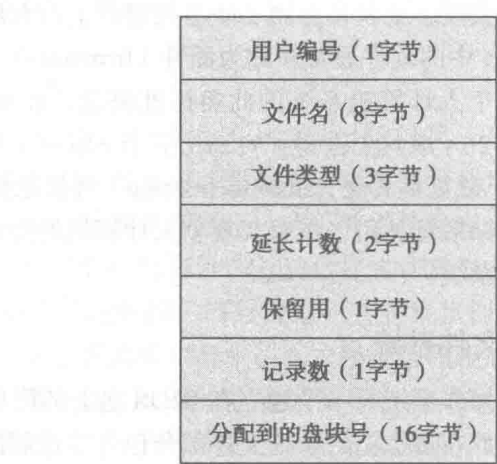


图 3-4 CP/M 文件目录项

1) 用户编号（user number）。这实际上是从 0 到 15 的分组编号，支持多个用户或组别共享磁盘并把他们的文件分别整理到同一组中。注意事实上并没有子目录，所有的文件都在

Ⓐ 在现代个人计算机中，类似的引导往往由硬盘而非软盘来完成。通常，硬盘由个人计算机制造商预装了操作系统。

Ⓑ 对于硬盘来说，当然会大出许多（译者注：目录所包含的目录项数会更多，每个目录项自身的字节数会更大）。

同一个目录下。分组编号会产生一种支持单级子目录的错觉。但实际上这些只是虚拟的子目录。

2) 文件名 (file name) 和文件类型 (file type)。它们可以被看作是一项, 具体由 1~8 个字符的文件名和 0~3 个字符的文件类型组成, 经常被称作 **8.3 格式文件名**。如果实际的文件名比“8.3 格式”短, 则短缺部分用空格填充。文件名中并不支持所有的字符。句点用于区隔文件名和类型 (如 MYFILE.DOC), 但并不存储在目录中, 所以文件名中不允许使用句点或空格。尽管操作系统附带的实用程序不允许非法文件名, 但 (应用程序用来创建、打开或重命名文件的) 相关操作系统调用实际上并没有检查文件名, 因此, 一个应用程序可以创建的文件, 却有可能无法被其他程序所访问。

[56]

3) 延长计数 (extent counter)。延长部分是由目录项管控的文件组成部分。如果一个文件占用的盘块数超过一个目录项所能指定的盘块数, 则必需为其配备额外的目录项。该文件的第一部分所对应的目录项的延长计数设置为零, 在此基础上, 文件的其余部分所对应的目录项的延长计数按顺序编号。于是, 大文件将拥有多个目录项, 它们具有相同的文件名, 但对应不同的文件延长计数和不同的分配指针集。因为文件可能被删除, 并且它们的目录项可能被重用, 所以目录中延长计数不一定有序。

4) 记录数 (number of records)。这实际上是延长部分所采用的 128 字节的记录的条数 (译者注: 每个文件的数据内容由分别对应于一个目录项的若干延长部分组成, 而每个延长部分又由分别对应后面所说的一个分配块的若干延长记录组成)。如果数据计数达到 080x (译者注: 即 128 字节, 一个扇区的大小), 则本延长记录已占满和用完, 故需要磁盘上的另一延长记录。文件长度被四舍五入到最接近的 128 字节, 因此应用程序必须知道在最后一记录中真正有多少数据。这导致了采用 Control-Z 字符来标记文本文件结束的惯例。

5) 分配映射 (allocation map)。这是包含文件数据的磁盘块的一组编号 (或指针)。每个指针 16 位 (即 2 字节), 共 8 个指针。每个指针指向磁盘上包含该文件对应组成部分的一个扇区。如果文件很小且所含扇区少于 8 个, 那么未使用的指针值将被设置为零。如果文件太大且 8 个扇区不足以容纳该文件的数据, 则需分配另一延长部分和填充另一目录项。在有些系统中, 设有 16 个指针, 每个指针 8 字节。这种不一致是制约 CP/M 推广和发展的主要问题之一。

数据存储区

数据存储区包含文件的数据块。当系统访问一个文件时, 用户或应用程序会提供文件名, 文件系统将搜索磁盘目录, 以确定是否有该文件名的文件存储在磁盘上。如果找到了拥有该文件名的目录项, 则从对应目录项中就可以找到存储相应文件的扇区的地址, 从而对文件数据进行访问。

[57]

注意: 真正的磁盘结构要复杂那么一丁点儿。如果你要统计上面描述的文件指针在所有文件中所包含的总数, 于是 $64 (\text{目录项}) * 8 (\text{文件指针} / \text{目录项})$ 共给出了 512 个扇区, 但在一张软盘上会有更多的扇区 ($77 * 26 = 2002$)。 (译者注: 这是单面软盘, 对于双面软盘, 扇区总数将在这个基础上翻倍, 即 4004 个扇区。) 这将导致大部分的软盘空间无法被利用。为此, 在现实当中, 文件指针并非指向一个单独的扇区, 而是指向把扇区组织到一起所形成的分配块 (allocation block) 即簇, 且分配块是组合在一起的连续的扇区。这些分配块的大小是由磁盘的大小决定的, 但在早期常见的软盘中, 分配块一般包括 8 个扇区, 即 1024 字节。因此, 每个目录项实际上可指向最多 8 个分别为 1024 字节的分配块。

如下为关于这种文件系统的局限性及一些评论：

- 目录中没有日期或时间。
- 没有项（译者注：即字段）可以明确标出文件的大小，故而文件大小必须基于对应目录项中所包含的指针数及可能的延长计数进行粗略的计算。
- 只有单一目录而无子目录，不过分组编号可造成支持一级子目录的假象。
- 一个文件必须整体全部存储在一张磁盘上。
- 如果目录已满（即全部被占用），则磁盘也被存满和全部被占用。因为目录具有固定的大小，所以一张软盘上只能存储 64 个甚至更少的文件。

评论：对 CP/M 的最大抱怨之一是 8.3 格式文件名，这是一件相对比较容易解决的事情。每个目录项（包括所存储的文件名）分别占有 32 字节。但其很容易就能够被扩展到 48 字节或 64 字节的目录项，从而支持 23.3 格式甚至 40.3 格式的名字（译者注：23.3 格式可能为著者笔误。对于 32 字节而言，除去文件名所占的 $8+3=11$ 字节，应还有其他部分对应的 21 字节；所以，对于 48 字节目录项，除其他部分的 21 字节，文件名应占 $27=24+3$ 字节，故应为 24.3 格式而非 23.3 格式；对于 64 字节目录项，除其他部分的 21 字节，文件名应占 $43=40+3$ 字节，故应为 40.3 格式）。不过，最初的设计非常简单，设计人员并不想把太多的磁盘空间用在目录上。权衡之下，对每个目录项所占的磁盘空间（和内存空间）进行了最小化设计处理。这对于软盘来说是特别重要的，因为文件内容和目录项加到一起往往多达几百 KB。

3.5 进程和内存管理

在后面我们即将学习的更为复杂的操作系统中，通常首先会论述进程和内存管理的问题，因为进程对应于用户想要让系统做的事情，故而相关问题是头等重要的。但在这个简单的操作系统中，进程管理和内存管理的功能极为有限，因为每次只有一道程序在执行。于是，硬件抽象和文件系统的问题便显得更为重要些。

尽管如此，即便是这些非常有限的功能，操作系统也必须针对进程和内存的若干问题加以处理。首先，我们讨论了程序执行过程的典型流程和基本环节。然后，我们讨论了命令处理。最后，我们讨论了内存管理和覆盖技术，后者可被运用于所创建程序的大小超出可用内存空间的情形。

58

3.5.1 应用程序的创建与执行

通常，应用程序是通过将程序语言指令输入文本编辑器中的方式来进行编写的。在此基础上，应用程序被编译或汇编处理（或者既被编译又被汇编）。最后，利用链接编辑器将其与库例程链接到一起，从而产生一个可以加载到内存和运行的程序映像（program image）文件。程序映像被赋予了许多名字，例如，可执行程序（program executable）、二进制程序（program binary）或可运行程序（runnable program）。经过几轮的调试之后，对应程序便准备就绪和可以使用了。

为了让一个程序开始运行，其可执行的二进制代码文件必须被加载到内存中。这个加载过程通常由 CCP 来完成。CCP 本身是一个提供了若干内置功能（built-in function）的应用程序。例如，“DIR”命令用于列出所有文件的目录，“ERA”命令用于删除一个文件或者一组文件。在完成相关工作的过程中，CCP 仅对 BDOS 进行了调用，而从未直接调用过 BIOS

或硬件。这样便使得 BDOS 更容易被移植到一个新的硬件系统上。当向 CCP 输入一个名称时，其首先要查看和确认是否为一个内置命令的名称。如果是，则执行对应命令。如果该名称不是一个内置命令的名称，那么 CCP 将试图在磁盘上寻找一个拥有该名称的可执行程序文件。如果能够找到，则会把对应文件的内容加载到内存中，从而开始运行相应程序。在程序头中有一个特殊的代码，可用于辨识和确定对应文件能够作为一个可执行程序。另外，用户输入的命令也可能是一个文本文件的名称，该文件由 CCP 读入和逐条执行的一串命令组成。在 CP/M 系统中，相关文件称作 **subfiles**，以标准扩展名“.sub”作为文件名的第二部分。这种类型的命令文件通常被称为**脚本文件**（script file）或**批处理文件**（batch file）。

在 CP/M 系统中，普通的应用程序总是加载到从地址 0x0100 开始的随机存取存储器 RAM 中。对所有的程序指定一个固定的加载地址，使得程序的开始地址将会事先被获悉，从而方便了编译器和链接器对可执行程序的创建。

通常情况下，程序需要额外的内存来存放静态数据，譬如像静态字符串之类的预定义的且大小固定的数据。同时，还需要栈机制来存放动态数据，如临时变量。预定义的静态数据加载到紧随程序二进制文件加载区域的内存中。紧随静态数据的其他内存空间保留用于存放其他数据。图 3-5 是关于一个正在执行的程序的各部分的一般的内存映射示意图。

初始化时，栈设立在高端内存中紧靠操作系统代码下面的位置。栈在内存中朝较低内存地址的方向增长（如图 3-5 所示）。加载完成后，操作系统调用对应程序的第一条指令，就好像其调用子程序一样。该程序一直执行，直到完成相应任务，此时控制权仅仅是返回到加载该程序的 CCP 中。CCP 可能还在内存中，但如果不在，就需要从磁盘重新加载到内存。当一道程序执行时，它可以在任何想要的时候使用所有可用的内存，除了包含有操作系统或操作系统组成部分的保留内存区域。

进程的执行，从开始一直到结束。其间，如果需要输入/输出，处理器将保持空闲直到其所需的输入/输出完成。例如，进程可能等待用户从键盘输入。对于不能完整加载到内存的大型程序而言，其程序代码通常被划分为可分别装入到内存的若干分段。当一个分段调用了对应代码不在内存的函数时，一种所谓“内存覆盖”的技术会被用来把新的代码分段装入内存，以替换当时存放在内存的另一代码分段。覆盖的实施交由应用程序具体负责，不过应用程序库通常会提供一些辅助支撑功能。我们将在第 3.5.3 节进一步讨论这项技术。

3.5.2 基于 CCP 的命令处理

在我们这个简单的操作系统中，CCP 跟任何其他程序非常相像。同时，CCP 或许比一些程序的结构设计得更好，因为它仅仅使用操作系统的系统调用而从不绕过操作系统。在其他的操作系统中，类似于 CCP 的模块有时也被称为**外壳程序**（shell）或**命令解释器**（command interpreter）。用户可以通过输入一条 CCP 的命令或一个可执行程序文件的名称，从而直接调用 CCP 的命令。这种命令解释器的另一个名称是**命令行接口**（command line interface），因为每条命令都被输入到屏幕的一行上，并当用户按下 <carriage return>

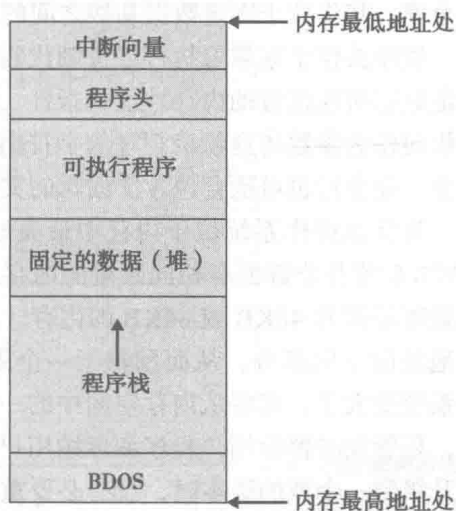


图 3-5 当执行一道程序时的典型的内存内容

或 <enter> 键（即回车键）时被提交给命令解释器。CCP 被链接加载到 BDOS 下的高内存区域。当一道程序运行完毕时，其将退出和把控制权返回到 BDOS，后者将会检查确认 CCP 是否完好无损地还在内存中。如果是，BDOS 将无须重新加载 CCP，而是会把控制权直接返回到 CCP。

许多用户更喜欢附加的功能或者不同的“外观和感觉”，例如菜单或图形。用更适合一个人喜好的外壳程序来取代 CCP，这种情况是相当普遍的。在许多操作系统中，用户可以从几个不同的外壳程序中进行选择。编写一个命令处理程序或外壳程序是一项很有意思的训练。但是，像许多程序一样，当你和其他人要使用该外壳程序时，其便需要增加一些新的和更复杂的功能，例如重新调用以往用过的命令或者将命令连接到一起使用的能力。

3.5.3 内存管理

正如我们在第 3.5.1 节所讨论的，CP/M 操作系统的基本的内存处理是非常简单的。所有的程序都加载到内存空间中一个固定的地址。程序被划分为两个部分：1) 程序的可执行代码；2) 固定的数据（fixed data，或称为静态数据，即 static data），如程序中的常量值和字符串等。

把这两部分从磁盘复制到内存的软件称为装入程序（loader，或称为加载程序），是 CCP 命令处理模块的组成部分。一个程序还需要一定的栈（stack）空间来存储临时变量、传送参数到所调用的子程序以及从那些子程序返回数据。栈被设立在内存中紧靠操作系统本身下方的最高位置。这便允许栈在内存中向下“增长”且不会与程序的数据发生“碰撞”或“冲突”，除非已经没有更多的可用内存。图 3-5 说明了这种内存结构。

但是，CP/M 并没有提供关于栈和固定数据之间发生冲突的检测机制。由于处理器并没有内存管理寄存器实现内存保护，所以相关事件往往会导致程序崩溃或引发奇怪的结果。此类内存重写的漏洞很难被发现和修复，在 CP/M 系统上经常会发生。

对于采用如 Pascal 或 C 等高级程序设计语言编写的程序来说，有一个大的内存池是可以被动态分配和回收的内存组成的，称为堆（heap）。堆是由装入程序从内存空间中划拨出来的，但由高级语言运行库中的例程负责管理。并非所有的程序都会使用堆。但如果存在一个堆，其将位于固定数据和栈之间的内存中。

程序头位于紧靠可执行二进制代码的前面的内存中。程序头包含分别指向栈所在位置和固定数据所在位置的内存地址的指针。当用户输入命令和为程序提供参数时，它还会包含一个指向作为参数传递给该程序的字符串的指针。例如，如果用户输入命令来运行一个文本编辑器，命令行也可能会包含要编辑的文件的名称。

为什么操作系统位于内存中最高地址的空间，而不是低地址的内存空间呢？这是因为 CP/M 系统并非都拥有相同数量的内存。一些计算机可能拥有 32KB 的内存，而其他的计算机则可能拥有 48KB 或 64KB 的内存。因此，操作系统被构建成（或配置为）占据内存的最高地址的空间部分，从而预留下一个固定的地址（常常是 0x0100）用于加载程序。如果操作系统变大了，其将从内存空间中的一个较低的地址开始，但不会为此强制任何程序修改地址，尽管此时留给用户程序和容纳用户程序运行的内存空间会变少。这便意味着，当操作系统升级到一个新的版本时，没有必要重新链接所有的应用程序。

3.5.4 覆盖

CP/M 系统的内存空间的最大值是受到对应处理器能够寻址的内存大小的约束的。最初, 相关内存空间最大值是 64KB, 但有些后来的处理器支持更多的内存, 还有一些计算机则添置了某些其他的硬件, 且相关硬件提供了经由程序控制可以被映射到内存空间的所谓内存“库存”。如果一道程序无法在可用空间容纳下, 会发生什么呢? 从最早的计算机开始以来, 这个问题就一直是个要害问题。

61

当然, 处理大量数据的程序可以把一些数据存放在磁盘上而不是内存中, 且在特定时间只是把所需要的数据装入内存。然而, 如果程序的二进制代码非常庞大, 又可以采取什么办法呢? 类似地, 这些程序也可以只是把正在处理所需的那些模块装入内存。这些模块或程序将会在内存中的相同位置相互“重叠”, 称之为覆盖。拥有大量代码的程序将会被划分为主程序模块和其他程序模块。主程序模块将常驻在内存空间。而其他程序模块则仅在需要时, 才会被加载到作为覆盖 (以替换另外的程序模块) 的内存空间中。典型的用于覆盖的候选程序包括一些大型的计算例程或错误处理例程。

程序设计人员必须能够识别并确定程序的哪些部分应当被组合到一起和形成一个覆盖。在设计覆盖时, 应该避免一个覆盖模块调用进入另外一个将会替代内存中该覆盖模块的覆盖模块 (译者注: 也就是说, 应当避免一个覆盖模块 A, 调用进入另外一个将会替代内存中该覆盖模块 A 的覆盖模块 B), 这是非常重要的。覆盖模块的真正加载由程序设计语言运行库来完成, 其利用 CP/M 系统调用来加载覆盖模块。程序设计人员应为编译程序标出每个覆盖可以对应程序的哪些部分 (即哪些函数和过程), 而编译程序据此生成可加载的覆盖代码。图 3-6 举例说明了相关概念。其间, 对应程序拥有一个主程序模块和三个覆盖模块。在任何特定时间, 同一覆盖只会有一个覆盖模块存放在内存中。

汇编程序是一个可以采用覆盖技术的程序实例。源程序通常会被读取两次。第一遍, 汇编程序构建一个符号表, 并为代码和数据分配空间。然后, 源程序被再次读入, 并发生实际的代码生成过程。此时, 汇编程序拥有足够的信息来生成指令, 并填写它们所引用的地址。显然, 针对源程序的这两遍处理不会直接引用彼此, 从而可以相互覆盖。在这个实例中, 经常至少会有其他两个可能的覆盖。一个覆盖是初始化阶段, 接收用户控制选项, 打开输入文件, 分配符号表, 等等。第二个覆盖则可能提供额外的关于文件的打印输出, 如附带有用户注释的生成代码的列表。尽管不采用覆盖技术, 整个汇编程序也可能运行在大型计算机上, 但在较小的计算机上, 如果没有覆盖技术, 就可能无法运行。此外, 可被处理的源程序的大小受到符号表存储空间的限制, 因而采用覆盖技术的汇编程序能够处理更大的程序。

62

3.5.5 进程及基本的多任务

即使是在内存容量有限而处理器运行速度缓慢的早期系统里, 用户也希望可以并行地做一些

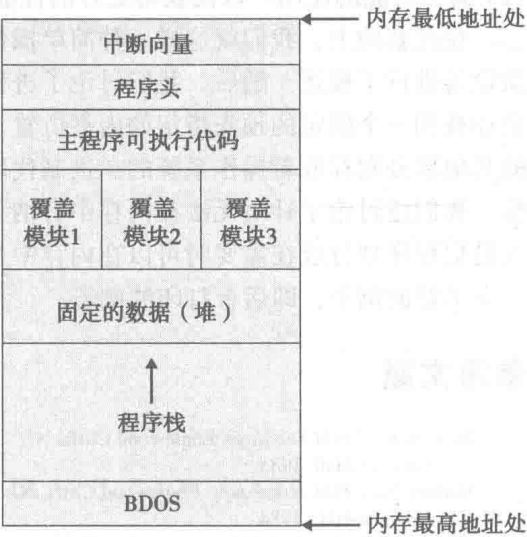


图 3-6 当执行一道程序时发生的内存覆盖

工作。一种很常见的请求是，在**前台进程**（fore-ground process）编辑（或玩游戏）的同时，可在**后台进程**（background process）打印一个文件的能力。这种同时允许另一前台程序运行的打印处理是一种非常普遍的请求方式。打印机是如此慢慢腾腾，为此，开启一项打印任务让计算机专心打印，然后起身离开 30 分钟或 1 小时是非常无聊的甚至令人厌烦，而且浪费了最宝贵的资源，即一个人的时间。特别当其间出了问题，而 30 分钟后用户返回时才发现打印机正在等待用户处理的情况下，显得更是如此。

CP/M 的解决方案是设立一个后台打印进程。一个小程序被加载到内存空间高地址部分紧靠操作系统下方的位置。该程序完成初始化后，就把控制权返回给 CCP，以允许另一个程序运行。当某用户想要在后台打印时，要打印的文件名便传递给后台打印处理程序。每当后台打印处理程序获得控制权的时候，便会打印文件的一小部分，也就一两行吧。通常情况下，每当前台进程执行系统调用或者可能通过设置定时器并造成前台进程中断的时候，后台进程就会获得控制权。

后台打印使计算机看起来似乎在同时做两件事情，即所谓的“多任务”。后台打印处理程序将只在后台专职打印事务，而不做其他任何事情。用户喜欢并行开展工作的理念，尤其是在输入和输出非常缓慢的情况下，譬如早期打印机的打印。我们将会看到，所有新型的操作系统，即使是在非常小的设备上的操作系统，也会提供某种形式的多任务处理机制。

3.6 小结

在这一章，我们介绍了一种简单的功能有限的操作系统的典型组成模块。我们的介绍建立在一种早期的个人计算机操作系统（即 CP/M）和早期个人计算机系统基本硬件的相关功能的基础之上。我们从描述简单操作系统的前身，即所谓的“监控程序”开始，讨论了它们是如何由于标准化的需要而演化成早期的操作系统的。然后，我们描述了采用这种类型操作系统的早期个人计算机系统的特征。接下来，我们讨论了在这样一种早期的操作系统中，输入/输出是如何进行管理的。我们认识到，应用程序经常会忽略操作系统所提供的标准化输入/输出功能的使用，以便获得更好的性能和更大的灵活性。

在此基础上，我们就这样一种简单操作系统的文件系统以及该文件系统所基于的标准磁盘设备进行了描述。随后，我们讨论了进程和内存管理。我们注意到，程序的二进制代码总是加载到一个固定的预先指定的内存位置，因为每次都只有一道程序存放在内存空间。内存的其他部分则存放着操作系统的二进制代码和固定的程序数据。栈区域保留用于存放动态数据。我们还讨论了针对无法在内存中容纳下的大型程序的覆盖技术。覆盖技术允许程序设计人员把程序划分成在需要时可以在内存中互相替换的若干分段。最后，我们讨论了多任务的一个早期的例子，即后台打印的例子。

参考文献

Barbier, K., *CP/M Solutions*. Englewood Cliffs, NJ: Prentice-Hall, 1985.

Barbier, K., *CP/M Techniques*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

Cortesi, D. E., *Inside CP/M: A Guide for Users and*

Programmers with CP/M-86 and MP/M2.

Austin, TX: Holt, Rinehart and Winston, 1982.

Dwyer, T. A., and M. Critchfield, *CP/M and the Personal Computer*. Reading, MA: Addison-Wesley, 1983.

网上资源

<http://www.digitalresearch.biz/> (CP/M创建者)

<http://www.seasip.demon.co.uk/Cpm/> (译者注: 圈外人士
建档站点对应站点地址已进行了更新调整, 即
<http://www.seasip.info/Cpm/index.html>)

<http://www.osdata.com/> (操作系统技术比较)

[http://www.pcworld.com/article/id,18693-page,
3-c,harddrives/article.html](http://www.pcworld.com/article/id,18693-page,3-c,harddrives/article.html) (硬盘特征)

习题

- 3.1 早期个人计算机的监控程序提供了哪些类型的有限的功能?
- 3.2 对于调用监控程序的相关参数进行了什么类型的错误检查? 可能的结果是什么?
- 3.3 在个人计算机时代, 有许许多多的小的新兴的硬件销售厂商, 而且他们所有用户都在吵嚷着要软件。在这种环境中, 导致真正操作系统开发的早期监控程序的特点是什么?
- 3.4 CP/M 操作系统和微软 DOS 操作系统设计运行的硬件系统的最重要的特征是什么? 由此导致的一些决策结果是什么?
- 3.5 早期程序的基本输入 / 输出需求并不算高。然而, 一些应用程序也有一些较为复杂的需求。在许多情况下, 由监控程序提供的功能要比 BIOS 代码的同样功能慢得多。当操作系统提供的功能遮挡了应用程序所需的功能或者操作系统所提供的功能慢得使应用程序的性能不可接受时, 应用程序设计人员是怎么做的? 由此产生了什么问题?
- 3.6 除了键盘和视频显示器, 早期操作系统还关注哪些非常重要的关键的输入 / 输出系统?
- 3.7 对于命令解释器而言, 用户输入的大多数命令是通过在磁盘上查找拥有该名称的程序加以执行来完成的。但也有一些命令并不会映射到磁盘的程序上。它们驻留在什么地方?
- 3.8 在软盘 (或硬盘) 上, 所有盘面上的各个磁头都会处于相应盘面上离磁盘外侧的同一相对位置的地方。当磁盘旋转时, 盘面的某一部分将会在各自己的磁头下通过。盘面的这一部分称为什么? 该部分被划分成更小的部分。那些部分是磁盘的最小的可寻址部分。那些部分称作什么?
- 3.9 CP/M 把软盘的内容划分为三个部分。这三个部分都是什么?
- 3.10 为什么 CP/M 操作系统驻留在高端内存而非低端内存?
- 3.11 覆盖是在系统内存很小的情况下使用的一种过时的技术, 在现代操作系统中已不再使用。这是真的还是假的?
- 3.12 尽管 CP/M 并不支持真正的应用程序多道处理, 但其却实实在在允许某种类型的后台处理。那是什么?

单用户多任务操作系统

在这一章，我们将讨论一种比在第3章中所讨论的操作系统更复杂的操作系统，而且是一种相当现代的操作系统。进一步说，我们来学习 Palm 公司开发的 Palm 操作系统™[⊖]。前一章所涵盖的 CP/M 系统，起初每次仅支持一道程序（或进程），在其主要使用期结束时，才扩展和引入了如后台打印的功能。相比之下，Palm 操作系统的设计从一开始就支持多个进程的同时运行。

67

我们在第4.1节中通过概要介绍 Palm 操作系统以及关于其内核基础的背景来开启本章的内容。在这类操作系统中还有好多其他的操作系统，其中塞班（Symbian™）公司开发的 EPOC 以及 Linux 和 Windows NT 的减缩版本最为出名。后者称为 Windows 手机版操作系统（即 Windows Mobile OS；原先为 Windows 掌上电脑操作系统，即 Windows Pocket PC OS）。在本章的结尾部分，我们将会介绍这些其他的操作系统以及在这个快速发展的领域中的一些最近的发展动态。Palm 操作系统是针对所谓个人数字助理（Personal Digital Assistant, PDA，或称为掌上电脑）或个人信息管理器（Personal Information Manager, PIM）的小型手持设备开发的，相关设备通常被单个用户用来记录和跟踪个人日程安排、通信录、待办事项列表，或者在行程途中访问电子邮件和万维网（World Wide Web）。目前，这些操作系统应用在拥有掌上电脑的很多相同功能的手机上。Palm 操作系统通常每次只是运行几道应用程序，并且能够在少量应用程序执行的同时并发运行一些操作系统进程。因此，它支持有限的并发执行任务。与第3章中所描述的单任务类型的操作系统相比，它提供了更多的功能，故而也将有助于示例性地说明现代版的简单操作系统。

在第4.2节，我们讨论使用 Palm 操作系统的手持式计算机的一些独特的硬件特性。这些与众不同的特性促成 Palm 操作系统设计中所做的一些决定及选择方案。在 CP/M 系统领域，我们在其几近结束的时候才见到其引入了同时在内存中存放多道程序，以提供诸如弹出式窗口和后台打印的功能。而 Palm 操作系统则支持更复杂的多道程序设计，为此，我们在4.3节讨论 Palm 操作系统中的应用程序进程和操作系统任务的调度。

当多道程序同时运行在一个系统中时，内存管理会变得更加复杂。程序不能再假设其可以使用所有的内存。当应用程序提出内存请求时，操作系统必须承担起为它们分配相应的内存区域的责任。因此，操作系统必须要监测每个应用程序所使用的内存，并在应用程序结束时对相应内存予以回收。鉴于此，在第4.4节进一步讨论内存管理。第4.5节介绍在 Palm 操作

⊖ 在本章中所描述的操作系统功能，涵盖 Palm 操作系统发行版 5 之前的各个版本。其发行版 5 是一种不同类型的操作系统，支持一种不同的处理器。我们认为，一类设备及相对应的大致在所描述层级上进行工作的操作系统将会继续存在，所以我们并没有修改相关素材来对应后来的版本。本章所涉及的功能可能是未来一段时间内学生在类似的低档操作系统中能够找到的比较具有代表性的功能。例如，伴随纳米技术的发展，这样的机器非常有可能，它们往往包含需要操作系统的计算机系统但却没有辅助存储器（secondary storage，或称为第二级存储器）。此外，就目前看来，旋转式的数据存储设备可能很快就会成为过去，而大多数的新的计算机将拥有巨大容量的主存和一些可移动的第三级存储器（tertiary storage），但是没有第二级存储器。因此，所有的操作系统可能在某一点上更像这个操作系统一样地进行工作。

系统中的文件的组织，而第 4.6 节则介绍 Palm 操作系统所提供的基本的输入 / 输出功能。

早期的掌上电脑在很大程度上都是基于文本的，尽管很多掌上电脑也具备了一些特殊的图标或一些小块的拥有图形化功能的屏幕部分。现在，此类设备则往往具备面向图形的显示功能。CP/M 是一种基于文本的操作系统，所以我们还在这一章中介绍了图形化用户界面（Graphical User Interface，GUI）的一些简单的特征。所有现代的操作系统都包括对图形化用户界面的支持，尽管对于操作系统本身而言，图形化用户界面并不是其所固有的。CP/M 系统上的程序都假定自己完全控制着系统，故而被设计成以确定的方式来进行交互。图形用户界面下工作的程序必须处理伴随程序主要流程而以异步方式发生的各类事件。为此，本章也引入了面向事件的程序设计。具体而言，第 4.7 节介绍显示子系统，第 4.8 节首先讨论面向事件的程序设计，然后介绍 Palm 操作系统中一个典型的应用程序的设计。最后，我们在第 4.9 节中对全章内容进行归纳总结。

在本书后面的部分，我们将介绍 Palm 操作系统及类似系统的一些更高级的功能。具体而言，第 20 章讨论 Palm 操作系统中的一些很有意思的子系统，并对需要交叉开发系统来开发此类受限环境的程序的基本道理进行解释，同时还对 Palm 操作系统的后来发行版本的一些进展进行介绍。

68

4.1 简单的多任务系统

Palm 操作系统是由 Palm 公司开发的，使用在他们的小型手持式计算机上。典型的手持式计算机装置如图 4-1 所示。该平台已经变得非常普遍。不少硬件厂商，包括腾跃公司（Handspring）、索尼（Sony）公司、美国国际商用机器公司（IBM），都已生产出了与相关技术一致的设备。同样的操作系统还被用在了多款手机上，包括 Treo 和三星 500（Samsung）。与大多数通用计算机或个人计算机相比，Palm 操作系统运行的环境具有一些不寻常的特征。这些特点促成了开发操作系统时所做的一些不寻常的决定。尽管如此，这些特点在即将面世的越来越多的系统中非常具有代表性，所以对于无论当前的还是未来的操作系统的设计者而言，这些特征实际上并不是可有可无的，而是相当重要。这些特点还限制了操作系统的设计目标，故而相关操作系统与第 3 章所介绍的单进程操作系统相比，只是复杂那么一丁点，具体总结如表 4-1 所示。

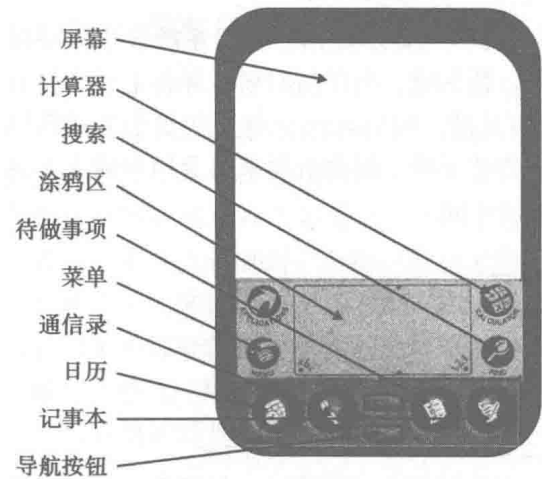


图 4-1 Palm Pilot 掌上电脑示意图

表 4-1 Palm 操作系统的与众不同的特征

实时操作系统任务，但并非实时的应用程序
全固态存储器
低功耗，以节省电池
单一窗口的图形化用户界面
多种文本输入选项
通过插件进行扩展

在这些不寻常的特点中，第一项特征源自于相关手持式计算机是它们之前的掌上电脑的成熟版本的事实。这些手持式计算机被设计成了赋予那些与用户进行交互的服务以最高的优先权，从而导致了操作系统实际上是建立在一个实时内核的上面，而该内核是由 Palm 公司从另一个供应商那里授权获得的^①。例如，这个实时内核允许系统支持在一小部分的液晶显示器（Liquid Crystal Display, LCD）屏幕上进行“写”操作的触控笔（stylus，或称为手写笔）的使用。该屏幕是触摸敏感的，触摸屏幕（最好是用手写笔）将导致一个中断把触摸到的屏幕位置的坐标传给一个用于跟踪触控笔在屏幕上移动的例程。操作系统试图实时地读取和解释笔迹，这在 Palm 操作系统中称为涂鸦式输入（graffiti input）^②。在操作系统处理这项实时任务时，也允许若干应用程序同时运行在相应机器上。支持多个应用程序及实时内核并发运行要求采用多任务（multitasking）或多道程序（multiprogramming）的系统设计。

在支持应用程序方面，Palm 操作系统设计如下：

- 读取邮件。
- 在通信簿中记录联系方式。
- 保持费用记录。
- 通过报警提醒来增强待做事项功能。
- 玩诸如数独（Sudoku）之类的简单游戏。
- 通过万维网访问信息。

Palm 操作系统不是特意要同时支持多个用户，也不是特意要成为一个网站服务器。因此，操作系统的实时和多任务的特点并没有通过应用程序编程接口（Application Programming Interface, API）暴露给应用程序编程设计人员。

这些系统的另一个独特的方面是，通常没有辅助存储器，也就是说，所有的系统内存均为主存储器（即电子类主内存）。这些手持式系统的有限的内存和处理器性能导致了内存管理的特殊设计及基本输入/输出操作的一些特殊处理。某些设备提供有插件功能，从而允许不同类型的卡（card）或模块（module）连接到相应设备上。相关卡可以是预装了特定应用程序的存储卡、全球定位系统（Global Positioning System, GPS）导航设备、数码相机甚至硬盘。然而，基本硬件并没有辅助存储器，所以操作系统的设计必须要反映这一点。关于辅助存储器的支持已被移植到主要的系统设计上，对此我们将在后面展开更详细地讨论。

Palm 操作系统支持面向用户的图形化用户界面方式的显示输出。鉴于系统较小的屏幕尺寸，所以界面编程设计往往会有一些特殊的考虑。特别地，在任何时候，屏幕上经常只有一个单一的窗口（window，或窗体，即 form）是可见的。有时可能会有一些更小的对话框或警报框显示在该窗口的前面。最后，这些系统支持若干种不同的机制来接受用户的文本输入，但系统试图在应用程序层面隐藏这些机制之间的不同。

4.2 Palm 操作系统运行环境及系统布局

在设计 Palm 操作系统时，需要考虑 Palm 设备的一些特点，具体包括：

- 基本内存为易变性随机访问存储器（RAM）。

① 该系统就是柯达产品有限公司（KODAK Products Limited）的 AMX™ 多任务执行系统。

② 2003 年，PalmSource 公司输掉了一场使用原创的涂鸦软件的官司。目前使用的该软件称作 Graffiti 2。我们采用简化的术语作为此项功能的通用名称。

- 通常没有辅助存储器。
- 小屏幕尺寸。
- 键盘并非标配。
- 处理器运行缓慢以降低电池消耗。

4.2.1 基本内存为易失性随机访问存储器

Palm 操作系统设计所支持的手持式计算机拥有一些与众不同的特征。首先,相关装置采用电池供电方式,故而硬件与操作系统的设计均须遵循和反映这一现实情况。如果系统几分钟未曾使用,其应当把自己置为睡眠模式(sleep mode)以使消耗很少的电量。此时处理器仍在运行,所以当用户按动按钮时,操作系统能够检测到。当然,此时的处理器运行得非常缓慢,且处于一个等待中断的小的循环之中。实际上,对内存的供电始终就没有停止过。即便当处理器和操作系统都已终止的时候,内存仍然处于加电状态。硬件拥有一股很小的电流来维持内存的内容。(为系统增加一些内存模块也是可能的,包括只读存储器(ROM)或可编程只读存储器(PROM),后者有时也称作闪存。不过,基本设计是假定所有的主存都是易失性的。)

4.2.2 没有辅助存储器

这些手持式系统的第二项与众不同的特征是,在最初的设计方案中,它们并不拥有任何辅助存储器,也就是说,没有磁盘、光盘、数字化视频光盘,也没有磁带驱动器。所有的数据和程序都保存在一个单一的地址空间里。该内存中有些是可以从计算机上取下的部件(或卡片)上的ROM。这便允许程序和数据库加载到相关部件上并根据要求插到相应的机器上。无论是在可移动的卡片上还是内嵌在机器上,所有的内存始终都是可见的,这样,所有的程序和所有的数据库就始终是可以直接访问到的。与Palm操作系统兼容的硬件的一些供应商还增添了一种独特的就像辅助存储器一样通过输入/输出命令来访问的内存类型。该内存并非主存地址空间的组成部分,故而需要特别的操作系统命令来进行访问。这类内存是可移动的,故将被用于从一个系统到另一个系统的信息迁移。其被设计成仿真磁盘驱动器的使用方式,所以在物理上也兼容其他的硬件系统。

71

4.2.3 小屏幕尺寸

下一项特征是把图形化用户界面呈现给用户的液晶显示器(Liquid Crystal Display, LCD)的类型。其系统功能与其他当前的系统——如采用了支持图形化用户界面的操作系统的个人计算机——所使用的阴极射线管(Cathode Ray Tube, CRT)或液晶显示器的屏幕相类似。主要的区别在于屏幕的大小(即屏幕的尺寸)。因为照字面意思来说,相关设备应当是被设计来要适合放到一个用户的手上的,屏幕显示自然受到了限制。对于大多数其他的图形化用户界面来讲,在屏幕上可以同时有多个窗口打开。于是这些窗口常常相互重叠,一些窗口的某些部分就被它们前面的其他窗口遮挡和隐藏起来了。为此,往往可能通过最大化一个窗口来填充(几乎)整个屏幕。

与其他的图形化用户界面相比,Palm操作系统的应用程序窗口会填满整个屏幕。相关应用程序仍旧可以使用下拉式菜单和对话框,但通常不会有其他的应用程序窗口被部分地隐藏在当前运行的应用程序的窗口的后面。

4.2.4 没有键盘

个人数字助理手持式系统的最后一个很有意思的方面是，它们最初没有键盘。有一些可用的外接拆卸式键盘，后来的一些模型才切切实实拥有了一个实际的键盘，但这并不是系统通常所假定的获取用户输入的方式。常见的输入方式是涂鸦式输入，正如在第 4.1 节中所讨论的那样。那才是被普遍接受的，因为大多数的个人数字助理的应用程序都不期望大量的输入。

图 4-2 给出了 Palm 操作系统的总体设计。紧靠硬件上面的是一个被称为硬件抽象层（Hardware Abstraction Layer, HAL）的软件层。其作用是把软件的其余部分同硬件设备的具体细节隔离开来，从而允许操作系统内核开发人员构建出一个很容易就能够移植到另一套硬件平台的操作系统。操作系统的内核位于硬件抽象层的上面。操作系统提供的许多服务并不是内核的组成部分，而是位于内核的上面。在系统服务层（其总是在内核之上的层次）的上方将会是可选的系统库例程，再上面会是应用程序库例程，而最上面，则是应用程序自身。



图 4-2 Palm 操作系统体系结构

72

4.3 进程调度

在 Palm 操作系统的多任务环境中，任何人都需要分清操作系统进程和应用程序进程。在这一节中，我们分别讨论了这两种类型的一些具体的进程，并阐明了 Palm 操作系统是如何处理和调度这些进程的。

4.3.1 处理涂鸦式输入——实时操作系统任务

正如在第 1 章中所提到的，有许多任务最好能够在操作系统中完成。把功能放在操作系统中有诸多的理由，但通常是因为许多应用程序都会用到它们。将对应功能放在操作系统里来实现，简化了应用程序设计人员的开发工作，同时保证所有的应用程序能够以相似的方式来执行功能，并降低了相关应用程序在对应部分发生错误的可能性。Palm 操作系统中关于此类任务的一个最好的例子是涂鸦式输入功能。Palm 操作系统所在计算机系统的显示往往是在一个触摸敏感的液晶显示面板上进行的。用户一般通过在这个屏幕上涂画字符来把数据输入掌上电脑。这是一类专门由操作系统来完成的任务，具体涉及手写笔跟踪（stylus tracking）和字符识别（character recognition）两项操作系统任务。

为了跟踪一支触控笔在液晶显示器屏幕的涂鸦区表面上的活动轨迹，处理器必须快速和反复检查确认触控笔的当前位置。相关跟踪是一项实时性的任务，因为系统需要能够保证其可以频繁地和足够迅捷地检查触控笔的位置以跟踪记录触控笔的运动。由于在这些设备上的处理器要比个人计算机或工作站上的处理器运行得更为缓慢，所以相关任务要更为麻烦些。当触控笔改变方向时，对应跟踪任务应能辨别出来，同时应把相关活动轨迹分解成一些小的向量，并将之传递给字符识别任务。一旦对触控笔的位置向量进行了分析和辨认，相应字符可以被识别出来。同样，字符识别任务也是由操作系统来完成的。每个应用程序的开发人员

都不想自己编程来实现一个手写识别器。事实上,这是 Palm 操作系统给市场带来的掌上电脑技术的进步之一。与触控笔跟踪任务相比,手写识别任务的处理要更为悠闲和宽松。当相应字符被识别出来以后,手写识别任务将把它们传送给相关的应用程序,后者必须将它们回显在液晶显示屏幕的适当位置上,以便用户获得字符输入的反馈信息,就像个人计算机上的键盘输入方式一样。

73

4.3.2 应用程序进程——任何时候只能有一道进程持有焦点

在大多数系统中,用户可以同时运行多道应用程序。但在 Palm 操作系统中,任何时候只会有一道用户应用程序明显地被看到正在运行中。尽管如此,大多数 Palm 应用程序并没有用户可以调用的“exit”函数。当用户选中一道新的应用程序时,任何正在运行的应用程序将会被操作系统从用户面前隐藏起来。为此,在 Palm 操作系统中,任何时候只会有一道正在运行的应用程序持有焦点(focus)——也就是说,掌控着屏幕窗口(screen window),并接受和显示输入。不过,其他的应用程序有时也可能运行,只是未持有焦点而已。此类功能的一个实例是文本搜索功能。如果用户执行文本搜索,Palm 操作系统会顺序调用那些已向操作系统说明提供有针对其自身数据库文件进行文本搜索功能的每一道应用程序。各应用程序将被请求在其数据库中搜索用户输入的待查字符串。但是,这些应用程序不会获得屏幕的控制权,而只是向操作系统汇报它们的结果。

在 Sync 应用程序(Sync application)中可找到关于正在运行但未持有屏幕焦点的任务的另一个实例。该应用程序用于实现手持式装置上的数据库文件与个人计算机上的数据库文件之间的同步。个人计算机运行着一道对应的同步处理程序,而两个系统之间采用某种串行通信链路进行通信。相关链接可能是红外方式、蓝牙方式或通用串行总线(Universal Serial Bus, USB)有线链接方式。尽管该应用程序通常持有焦点,但同步处理操作执行的同时往往没有用户输入。当然,有时用户可能想要在同步处理完成之前终止同步处理操作。能够满足此类要求的一种解决方案是把 Sync 应用程序设计成“发送一块数据然后检查屏幕是否发生触控笔敲击”的循环框架。不过,如果这样,将会降低串行通信的效率,并会延迟对触控笔敲击的响应。鉴于此,Palm 系统的 Sync 应用程序采用了两个任务并举的解决方案,即通过一个实时任务基于中断进行屏幕触碰事件的响应,而同步处理应用程序则全力专注于通信任务。

4.3.3 典型的用户应用程序

大多数的 Palm 操作系统应用程序主要涉及数据库和图形化用户界面接口,并用来设计和实现信息相关的组织。典型的应用程序包含有待办事项清单、地址和联系人信息、预约日程表以及各式各样的闹铃。为此,它们并不直接涉及实时任务。正如前面所描述的那样,操作系统利用实时任务来处理触控笔输入操作。应用程序本身只是输入和显示诸如用户预约之类事务相关的信息。所以,普通的用户应用程序并不需要启动额外的任务,就像 Sync 系统应用程序一样。各应用程序的主干部分是一个所谓事件循环(event loop)的循环。操作系统“启动”应用程序。应用程序检查确认这是否是其第一次运行。如果是第一次运行,则其将初始化所维护的所有数据库。然后,进入等待事件的循环中。大多数事件是诸如涂鸦式输入所做的字符识别或菜单列表选定菜单条之类的活动。

有一些独特的系统事件,譬如,通知所有的应用程序“系统即将进入睡眠模式”。另外一种常见的事件类型是“appStopEvent”。正如之前所提到的,当用户选定一个应用程序运

74

行时，该应用程序将成为活跃的应用程序，同时操作系统将强迫当前运行的应用程序停下来。在不同的环境中，另外的操作系统不会仅仅因为一道应用程序未持有焦点就把它停下来。如果用户要切换回该应用程序，重新启动该应用程序将需要太多的输入/输出和处理器处理。然而，在 Palm 手持式系统上，无论是程序还是文件，都始终存放在主存空间中，所以并不需要完成诸如给程序分配内存、从磁盘驱动器读取可执行模块并打开相应文件等各项任务。如果用户重新选中一道已经关停的应用程序，该应用程序所做的全部事情就是，认识到其文件已经初始化，并进入关于检查其需要处理的事件队列的循环过程之中。为此，对于只是等待用户从菜单中或通过图形化用户界面选择一些功能的典型的应用程序而言，关停可能并不意味着什么。不过，例如，对于正在运行的用户跟计算机对抗的游戏类程序，如果用户要切换到另一道应用程序，则相应游戏程序将可能暂停相关操作。

4.3.4 真正的调度程序开始成形

就 Palm 操作系统所使用的实际的进程调度程序来说，其是一种抢占式多任务调度程序。这意味着该调度程序已经准备好要运行多道任务和根据需要在这些任务之间进行切换，从而满足系统的需求。不同类型的任务具有不同的优先级，操作系统调度程序将动态地确定哪个任务是最重要的，进而中断一个不太重要的任务和转向运行一个更为重要的任务。中断一个任务来运行另一个任务被称为抢占（preemption）。通过把处理器从不那么重要的任务那里抢走，从而达到使更重要任务可以率先执行的目的。操作系统的各种类型的处理器调度程序将在第 9 章展开进一步详细的讨论。

4.4 内存管理

由于在 Palm 系统中有许多进程共享着主存，所以操作系统必须提供大量的内存管理功能。第一项任务就是查看各个进程不要访问它们自身分配到的内存以外的任何位置，同时还必须要监测当前未被使用的内存的情况。

4.4.1 内存基础知识

在 Palm 系统中的内存访问采用 32 位地址，总共支持 4GB（译者注：实际应为 $4 \times 1024 \times 1024 \times 1024$ 字节，约为 4 千兆字节）的地址空间。应用程序所见到的内存视图反映了这一点，而实际的物理内存是在一块或多块存储卡（即内存条）上。每块存储卡对应最少 256MB（译者注：实际应为 $256 \times 1024 \times 1024$ 字节，约为 256 兆字节）的逻辑地址空间。存储卡通常是可更换的，所以系统的内存总量可以升级和扩大。尽管最初的硬件设计仅支持一块存储卡，但比较新的系统则支持多块存储卡。存储卡可能包含如下三种不同类型的内存：

- 只读存储器（Read-Only Memory, ROM）。
- 可编程只读存储器（Programmable Read-Only Memory, PROM；也称为闪存，即 flash memory）或非易失性随机访问存储器（Nonvolatile RAM, NVRAM）。
- 随机访问存储器（Random Access Memory, RAM）。

所有的存储卡都至少包含一些 RAM；而存储卡上是否出现其他两种类型的内存则取决于存储卡。起初，操作系统及整个初始的应用程序集都包含在 ROM 中，但现在它们往往存放在 PROM 中，从而方便它们的更新升级。其他的应用程序则可以安装在 PROM 或 RAM 上。

从逻辑上来说，RAM 划分为两个部分：1）一部分 RAM 当作是易失性的，称为动态型

RAM；2）另一部分 RAM 当作是非易失性的[⊖]（NVRAM），称为存储型 RAM。

如果存储卡上有可编程只读存储器，则总被认为是存储型随机访问存储器，因为其实际上是非易失性的。动态型随机访问存储器的使用跟传统的随机访问存储器在大多数计算机系统中的应用一样。当系统关闭（即转入低功耗睡眠模式）时，整个随机访问存储器的内容被保存下来。但是，当系统开启或被引导（booted）时，动态型随机访问存储器部分的内容会由操作系统重新设置。存储型随机访问存储器部分按大多数系统中的磁盘驱动器使用的相同的方式来进行使用——用于存放那些旨在长久保留的持久性数据（persistent data；即文件或数据库）。存储型随机访问存储器也可以存放操作系统的扩展部分（可能是修复之类的程序）以及其他的应用程序。

既然存储卡是可更换的，就需要有一种机制能够把存储型随机访问存储器中所包含的内容保存下来。具体方法是，先利用 Sync 应用程序把存储型随机访问存储器的内容同步到个人计算机上，然后更换存储卡，再把个人计算机上的相应内容同步回 Palm 系统上。采用这种方法时，个人计算机充当了存储卡内容的备份设备。同样，我们也可以考虑把 Palm 系统作为移动设备（mobile device），从而来缓存通常驻留在个人计算机上的部分用户文件和数据库的副本。

4.4.2 内存分配

内存被 Palm 操作系统作为堆[⊖]（heap）来进行管理——也就是说，内存分片由操作系统进行分配和跟踪；当应用程序运行时，在堆内进行访问；最后，由程序释放，并被操作系统归还到可用的内存池中。这些内存分片在 Palm 操作系统中被称为内存块（memory chunk）。至少有三个堆，即每类内存（包括 ROM、动态型 RAM 和存储型 RAM）分别各作为一个堆。在较新版本的 Palm 操作系统中，一些这样的内存区可能被划分成两个以上的堆。当一个应用程序提出请求时，操作系统就在各个堆中分配内存块。这些内存块可以是非零的、按两字节递增的、最多稍小于 64KB（译者注：64×1024 字节，约为 64 千字节）的任何大小。内存块可以按任何次序归还给操作系统，并可通过操作系统服务调用来变小或变大。

内存块随机进行分配和释放，且大小可以改变。如果要变大，那么相关的内存块可能不得不移到堆中的另外一个地方。最后，这一过程会陷入一种所谓外部碎片化（external fragmentation）的情形。该术语描述的是这样一种情形，虽然存在空闲可用的内存块，而且空闲内存总量足以满足一个新的申请，但是由于即便是其中最大的空闲块也因过小而无法满应对应申请，所以相关内存申请还是不能直接得到满足。具体如图 4-3 所示。

尽管这个堆中，总共有 96 字节的空闲空间，但由于空闲空间被碎片化了，所以我们无法满足任何大于 16 字节的内存块的分配。

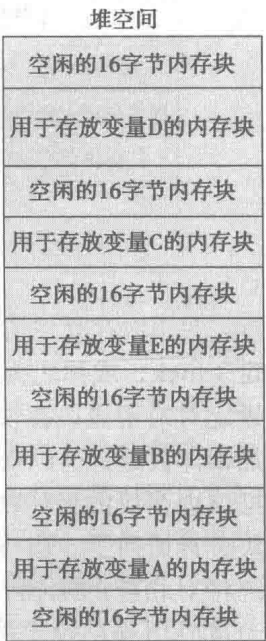


图 4-3 外部碎片化

⊖ 非易失性内存存在掉电情况下不会丢失其内容。
⊖ 堆是指一种数据结构，其中的内存按需进行分配，且没有特定顺序或次序。

当这种情况发生时，操作系统将尝试通过移动当前在用的内存块来形成连续的空闲空间。碎片化空间的这种重新组织称为**紧凑（compaction）**。内存重新组织可能带来一个潜在的问题：一个应用程序已经分配获得相关内存块并拥有诸多指向这些内存块的指针。如果操作系统准备移动相关数据，则必须保证该应用程序仍能够访问到那些数据。

考虑到内存空间中相关内存块的这种移动，所以被占用内存块应以一种受控的方式进行访问。首先，数据是被相应代码间接地访问而非直接访问的。如果那样，操作系统就可以在堆中移动数据，而且进程仍能够通过指针访问到相应数据。通过所谓的**主控指针表（Master Pointer Table, MPT）**中的一个表项来对堆中的每个内存块进行标记。主控指针表本身是对应堆的开始位置的一个随机访问存储器类型的内存块。当一个内存块被分配时，应用程序并未被赋予指向对应内存块的直接的指针，而是获得了一个**主控内存块指针（Master Chunk Pointer, MCP）**。该指针其实是主控指针表中对应于该内存块的指针的偏移地址，如图 4-4 所示。

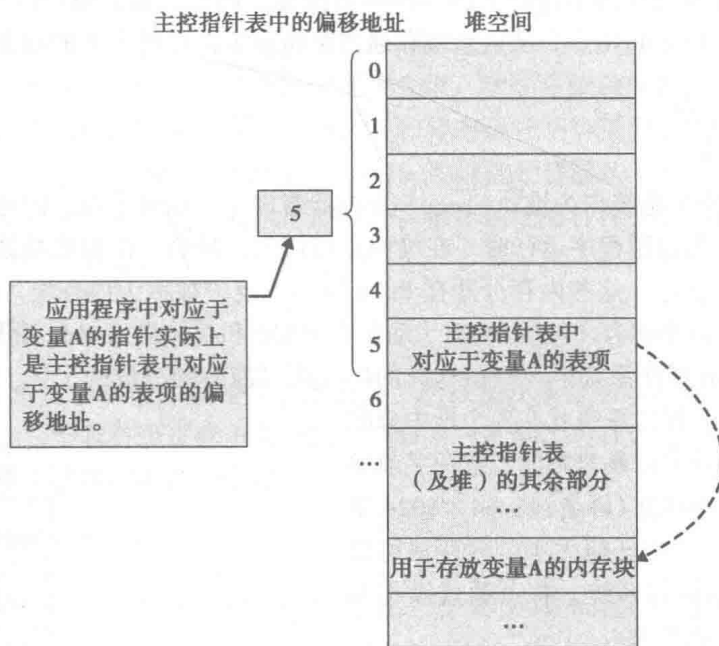


图 4-4 主控指针表

关于内存受控访问的第二个方面是指，应用程序在使用一个内存块之前必须先将其**锁定（lock，或称之为加锁）**。当应用程序想要使用内存块中的数据时，其调用操作系统来锁定对应于该内存块的主控内存块指针。鉴于操作系统为每个内存块维护着一个相应的加锁计数器，所以操作系统将就该内存块的加锁计数器做加 1 操作（最大值为 16），然后把该内存块的当前物理地址返回给应用程序。于是，应用程序就能够根据需要对该内存块进行访问了。当应用程序结束内存块的使用时，其应当为该内存块解锁，这时操作系统将对相应的加锁计数器做减 1 操作。当操作系统需要执行紧凑操作时，其并不会移动被应用程序锁定的那些内存块，因为加锁意味着有关的应用程序正在使用着相应的数据。

每个控制着一个特定的堆的主控指针表还包含有一个指针，以指向下一个可能的主控指针表。如果第一个主控指针表填满了，将会从堆中分配得到第二个主控指针表，同时第一

个主控指针表将会指向第二个主控指针表。图 4-5 和图 4-6 举例说明了相关概念[⊖]。

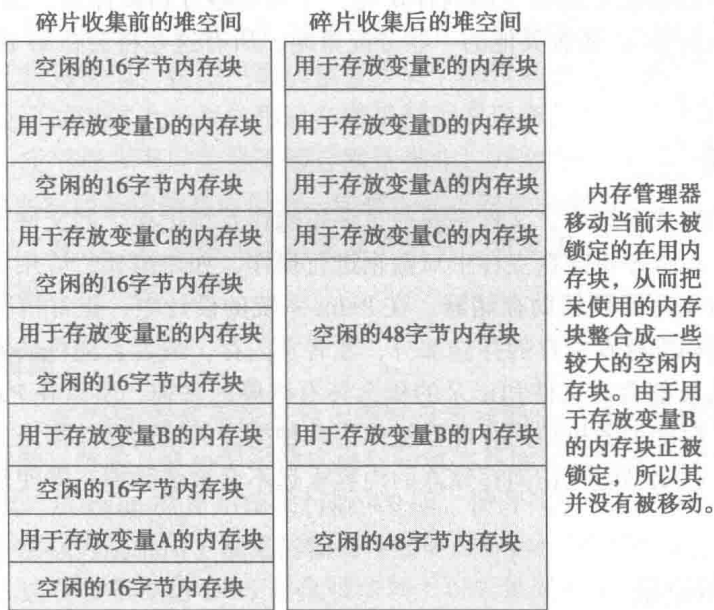


图 4-5 碎片收集

4.4.3 不可移动的内存块

一些内存块是不能被移动的，例如程序代码。不可移动的内存块是从堆的高端（高端内存地址）起进行分配的，而可移动的内存块则是从堆的前端（低端内存地址）起进行分配的。在主控指针表中无须为不可移动的内存块设立表项，因为主控指针表的唯一目的就是方便紧凑过程中内存块的移动操作。为保持一致，甚至连只读存储器都要通过“内存块表”来进行访问。这样就可以支持一个应用程序在随机访问存储器中进行调试后，不经修改就能够移动到只读存储器中。鉴于只读存储器中的代码按照定义是不可被移动的，所以在主控指针表中并不会为只读存储器中的堆设立相应的主控内存块指针。

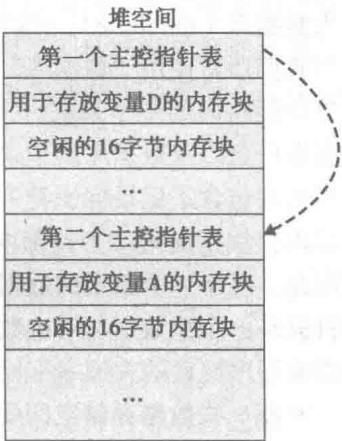


图 4-6 主控指针表链接

4.4.4 空闲空间监测

在堆最初由操作系统创建的时候，内存管理软件会创建空的主控指针表。如前所述，可移动的内存块是从堆的前端起进行分配的，而不可移动的内存块是从堆的后端起进行分配的。两者之间的区域被认为是空闲的内存。当应用程序有一些内存块不再需要的时候，它们就会调用操作系统来释放对应的内存块。被释放的内存块被标记为是空闲的，并会在需要的情况下被再次分配给相应的应用程序。如果所提出的是一个关于较小内存容量的请求，那么一个较大的内存块将被分割成两个部分，一部分分配给对应数据，而另一部分标记为空闲（未使用）。这便造成了堆的碎片化。但是，操作系统是如何决定针对哪个空闲的内存块来进

77
79

⊖ 这种机制看起来十分复杂，而且也确实复杂。然而，这是今天的许多操作系统中所使用的典型的内存访问控制机制，所以值得详细地审视和研究。

行切分的呢？它会选择一个满足要求的最小的空闲内存块呢？还是选择查找过程中发现的第一个足够大和可以满足要求的空闲内存块呢？相关策略分别被称为“最佳适应分配策略”和“首次适应分配策略”，还有其他的一些分配策略，所有这些将会在第 10 章中展开进一步讨论。

4.5 文件支持

在较为传统的操作系统中，文件系统通过调用操作系统把单个的文件记录从辅助存储器读取到主存中。应用程序将会在主存中对数据进行操作，如果需要，应用程序还会再次通过调用操作系统把数据写回到辅助存储器。在 Palm 系统的设计中，正常情况下并没有辅助存储器。所有的数据都保存在主存的存储部分，或者是闪存，或者是随机访问存储器。考虑到大多数程序设计人员都会对文件和记录的概念怀有浓厚的兴趣，所以在 Palm 操作系统中也保持了这种取向。存储型随机访问存储器被作为某种类型的辅助存储器来进行使用。正如前面所提到的那样，存储型随机访问存储器的内容永远不会被清除掉，即使是在系统被关闭的情况下。

4.5.1 数据库和记录

数据以记录（record）方式进行保存。例如，一条记录可能对应一部通讯簿中的一个联系人的联系信息。每条记录存放在一个内存块中。相关内存块被聚集到一起而形成的集合称为数据库（database）。（这些数据库是指在大多数操作系统中的所谓的“文件”。它们并不是当我们通常使用“数据库”这个词的时候所指的那个意思，即拥有包括自动对数据进行索引等功能在内的一个系统。）每个数据库都有一个首部（header）。该首部包含了数据库的一些基本信息以及数据库的记录列表。此记录列表实际上只是各记录的唯一标识符的列表。如果最初的包含了记录标识符列表的内存块已满，则会分配另一个首部内存块，同时将第一个首部内存块指向第二个首部内存块。相关标识符只是在单个的内存卡的地址空间内是唯一的，因此，对于一个给定的数据库的所有记录都必须存放在一个单一的内存卡中。虽然记录标识符只是一个整数，与对应数据并无任何关系，但是也有可能在每个数据库记录中创建一个可以被程序搜索的关键字字段。

在一些数据存储空间受限的（非 Palm 的）系统中，数据可能被压缩以节省空间。不过，由于 Palm 操作系统平台的处理器的处理能力不算太强，所以相关信息往往并没有被存储成压缩格式。当辅助存储器是在如磁盘之类的一个旋转式存储器的情况下，应用程序提出记录请求的时间和硬件可以访问数据的时间之间往往会存在一定的时延（time lapse，或 latency，即延迟）。该延迟时间常常可以抵消完成压缩操作和解压缩操作所需的时间。鉴于在 Palm 操作系统平台上通常没有旋转式存储器，数据获取几乎不需要时间，所以用于压缩处理的任何时间将对用户是可见的。为此，正常情况下，Palm 操作系统不会使用压缩处理。

4.5.2 资源对象

在图形化用户界面中，包含有诸如按钮、文本框、滑动条控件等许多出现在屏幕上的元素。Palm 操作系统把图形化用户界面的各式各样的元素定义为对象（object），称之为资源（resource）。这些资源不是传统意义上的对象，而只是数据结构。每个资源都有一个特定的结构，以便操作系统可以某些默认的方式对其进行处理。例如，如果一个应用程序要显示

关于一条记录被删除的确认式警示，它仅仅需要定义相应警报，然后调用操作系统来显示它即可。图 4-7 显示了这样一个警报框。当显示警报时，操作系统将会保存即将显示对应警报的下面一层的那个窗口并更新窗体上的窗口，以使用户看到警报。在用户确认该警报后，操作系统将把所保存的窗口还原回窗体上，并把用户在警报框上所选择的按钮告诉应用程序。应用程序可以始终不理睬默认操作，并触发一些特殊的活动。这些资源就像数据库记录一样被保存在内存块中，并被操作系统进行了标记，以方便操作系统弄清楚每个对象表示的是什么类型的资源。

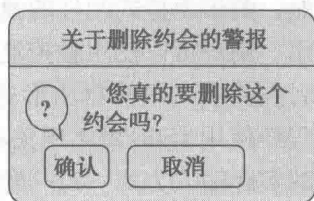


图 4-7 一个警报框窗体

4.5.3 辅助存储器

我们曾提到过，在 Palm 操作系统平台上通常没有辅助存储器。从大多数应用程序的观点来看，这是正确的。然而，在小型手持式设备领域的其他方面的发展却带来了关于更通用的存储机制的要求。从 Palm 操作系统发行版 4.0 起，便开始引入一种不同类别的内存设备的支持。相关设备被认为是采用了常见的辅助存储器设备的比较典型的组织方式。一种流行的模型应运而生，这些设备模仿 DOS 磁盘驱动器上可以见到的文件系统的方式来进行初始化。相关部件的期望的用途是，先被另一个设备（如一台个人计算机）写入信息，然后插入 Palm 操作系统的硬件装置上来开展后续的访问。用户可能在个人计算机上保存了好多文件，而把个别文件加载到内存部件上，以便于今后插入 Palm 系统上进行访问。为了使个人计算机无须特别的软件来访问常规的 Palm 系统的内存部件，于是采用了一种已经得到许多操作系统支持的文件组织方式。考虑到微软操作系统无处不在的客观现实情况，几乎所有现在的操作系统都支持可拆卸的辅助存储器设备所用到的那些文件格式。

81

4.6 基本输入 / 输出

隐藏硬件细节

Palm 操作系统贯彻这样的设计理念，即让相应系统在程序设计人员看来尽可能多地与传统的计算机系统别无两样。这方面的一个很好的例子是关于用户输入的处理。正常情况下，一个操作系统会隐藏用户键盘输入的许多细节。一般来说，至少会有两个层级的抽象：

1) 一些程序希望看到每一次按键操作。像 UNIX 文本编辑程序 vi 之类的全屏幕文本编辑器就是很好的例证。此类应用程序解释每一次按键操作，从而使其仅仅通过有限的几个按键就能够实现非常强大的编辑命令。这被称为原生接口 (raw interface)。

2) 第二个抽象层级可用于只要求按读取一整行来进行输入的应用程序。伴随用户输入每一行，操作系统会提供相应行的各式各样的编辑操作，包括字符或字符串的插入或删除、前一行的复制、退格、(在打错字母上的) 重打，等等。对应程序只会见到完整的输入行。这被称为熟式接口 (cooked interface)。

习惯用 C 编写程序的程序设计人员都知道，“熟式”的键盘接口体现为 `stdin` (标准输入) 功能。C 库通常还为打印机输出提供了一个所谓 `stdout` (标准输出) 的“熟式”风格的接口，为错误报告提供了一个相类似的所谓 `stderr` (标准错误) 的输出接口。起先，相关的输出流被设计成直接传送到硬拷贝打印机上，但后来，有关实现变成了常常把 `stdout` 定向到终端屏

幕而非真实的打印机上。Palm 操作系统的相关功能与此类似，提供了所有这三种接口。当然，关于手持式硬件的独特之处在于其正常情况下并没有键盘，这一点增强了相关抽象的效用。用户可以使用触控笔在手持式显示器屏幕上选择字符图标，就像键盘输入方式一样，或者在涂鸦区书写自由形式的字符。操作系统隐藏了所有这些细节，允许一个 C 语言编写的程序使用 `stdin` 而不用关心那些背后的实现细节，接受一个完整的输入行而不用担心其具体的实际输入方式。当一套真实的键盘连接到手持式装置上时，其将允许用户通过按键来输入命令，而且应用程序永远不会意识到其间的不同。

4.7 显示管理

4.7.1 相应硬件

82 标配显示器是触摸敏感式液晶面板，分辨率为 160×160 像素（图片元素或点），高分辨率显示器可达 480×320 像素。最初的模型只是白底黑字，但后来的模型可以支持四级灰度显示。更新的模型则能够显示彩色，具体为 2 种、4 种、64 种、256 种或 65 K 种颜色。就像早期的个人计算机一样，其屏幕是直接从内存提取相关内容来进行刷新的，而并非一件必须通过输入/输出指令来进行写入的设备。考虑到实际的显示器在不断地更新换代，所以强烈建议应用程序把硬件细节抛给操作系统去进行处理，而通过使用标准的系统调用来访问显示器。这是操作系统所做的一种代表性的抽象，目的是使应用程序无须处理硬件细节，从而具有更好的可移植性。

4.7.2 高级图形化用户界面元素

Palm 操作系统拥有一套基于窗体（form）概念的图形化用户界面。这些窗体类似于支持图形化用户界面的其他操作系统中的所谓的窗口（window），不同之处在于它们常常会填满整个屏幕。一个窗体通常是指应用程序数据的某一部分的视图。例如，地址簿应用程序可能有一个窗体用于查看地址列表，还有一个窗体用于编辑单个地址，等等。操作系统还支持一个称为窗口的元素，不过，这时的术语窗口指的是一个可以通过系统的绘图功能在上面进行操作的对象。有的窗口可能并不是窗体，而是用来创建对话框等。但所有的窗体都是窗口。在大多数情况下，应用程序并不会直接在窗口上进行绘制。所有的操作将基于按钮、菜单等图形化用户界面元素，或应用程序所做的系统调用来进行实施。进一步说，操作系统应支持按钮的绘制及用户点击按钮操作的具体处理；而应用程序仅仅需要在按钮上定义标签，并告诉操作系统在窗体上放置按钮的具体位置以及当用户触摸屏幕上的按钮时应提供给该应用程序的特定数字编码。该数字编码将作为事件提交给应用程序。另外，如果应用程序想支持动画或想定义自己的操作系统所没有提供的其他图形化用户界面元素时，其唯有使用低级的绘图功能。在 Palm 操作系统中，这些应用程序特有的图形化用户界面元素被称为小工具（gadget）或对象（object）。（在其他操作系统中，它们又常常被称为小部件，即 widget。）它们并不对应程序设计中所用的术语“对象”，而只是指可能拥有与事件相关联的特定子程序的数据结构。相关事件包括完成某区域绘制、点击屏幕上某窗体的按钮等。

4.7.3 特殊的窗体类型

有两种特殊类型的窗体无须填满整个屏幕，警报框（alert box）是其中之一。一种常见

的警报框可能是关于一条记录删除的确认，如图 4-7 所示。警报框由应用程序控制显示。应用程序要求相应警报框一直持有焦点，除非得到用户对该警报框的确认。这种窗体被称为**模态窗体**（modal form）。在某些情况下，只有一个唯一的按钮，且用户必须点击该按钮才可以实现对相应警报框的确认。在如图 4-7 所示的例子有两个按钮，用户可以通过触控笔具体选择其中的一个。之后，操作系统就会从屏幕上移除该警报框。这种窗体的另外一个特殊的地方是，不要求应用程序特别地来创建具体的窗体——而仅仅需要对定义了（出现在警报框及按钮上的）文本的相关结构进行设置，并请求操作系统来创建警报框。操作系统将处理包括点击按钮之类的所有事件。

第二种特殊的窗体是**进度对话框**（progress dialog）。这种窗体类似于警报框，但具有较强的动态性。其主要用于当一个应用程序正在完成比较耗时的处理（如文件传输等）的场合。应用程序可以通过一个相对独立的调用来改变当前正在显示的具体内容。一般情况下，会是关于应用程序进度的一个指示器。如果某应用程序正在发送一个 100KB 大小的文件且已经发送了 50KB，则可能会画一根长条，且着色该长条 50% 的部分。这便可以清楚明了地提醒用户还需多长时间才能完成相应操作。其间，通常有一个按钮可供用户点击，譬如关于取消操作的按钮。监控按钮点击是操作系统为应用程序提供的实时任务类服务之一，其不会中断应用程序的正常流程，但却能够对按钮操作事件提供及时的响应。这将有助于把应用程序从不得不时时检查按钮点击的困局中解脱出来，进而专心致力于正常的处理事务。

83

4.7.4 低级图形化用户界面控件

Palm 操作系统的图形化用户界面的控件不是传统意义上的对象。它们没有方法或属性，而仅仅是数据结构。对于给定类型的控件而言，有各种不同的操作系统调用可以用来触发它们的显示。当用户触摸到屏幕上的一个控件时，将会生成一个事件并被传递给应用程序。该应用程序将会接收事件并执行相应的代码。

表 4-2 给出了 Palm 操作系统所支持的控件、相关例子以及关于某些控件的一些具体说明信息。

表 4-2 Palm 操作系统所支持的控件

系统定义的控件	
控 件 名 称	详 细 信 息
按钮	触发某项功能或调用某个函数（如“显示”）
按键式按钮	如单选按钮
关于选择器的触发器	用于打开一种专门的对话框（如用于输入日期的对话框）
增量式箭头	用于改变对应关联控件里的数值
复选框	真 / 假——打开 / 关闭
弹出式列表	由弹出式触发器激活和打开
弹出式触发器	用于打开弹出式列表（标记为向下的三角形▼）
列表	下拉式列表（如菜单上的下拉式列表）
菜单	用于访问不常使用的功能
文本框	基本的数据输入框
滚动条	当数据溢出窗体显示区域时使用

4.8 事件驱动的程序

在 Palm 操作系统上，大多数应用程序都编写成交互执行方式。它们一般不会像主机上的工资管理应用程序那样处理批量数据，也不会像服务器那样对完整的独立请求进行响应，而是专注于用户的即时交互式输入。因此，这些应用程序是以一种特殊的方式组织起来的。当一个 Palm 应用程序运行时，其首先初始化相关文件（如果有），然后就进入一个检查事件的循环。在该循环中，它会检查和处理由操作系统传给它的各类事件。具体例子如图 4-8 所示。

```
static void EventLoop(void)
{
    UInt16 error;
    EventType event;
    do
    {
        EvtGetEvent(&event, evtWaitForever);
        PreprocessEvent(&event);
        if(! SysHandleEvent (&event))
            if(! MenuHandleEvent(NULL,&event,&error))
                if(! ApplicationHandleEvent(&event))
                    FrmDispatchEvent(&event);
        #if EMULATION_LEVEL != EMULATION_NONE
            ECApptDBValidate (ApptDB);
        #endif
    }
    while (event.eType != appStopEvent);
}
```

图 4-8 一个事件驱动的程序的主循环

如果程序没有事件需要处理，就会告诉操作系统要等待事件的发生。当下一个事件发生时，操作系统将把相关信息作为检查各类事件的一个函数调用的返回传递给该应用程序。举例来说，假设某用户已经启动了应用程序来准备完成某项特定的任务——比如说是在通讯簿文件中查找一个电话号码。该程序将无事可做而只有等待，直到用户交给它一项具体的任务。用户使用菜单和窗体里的其他控件来告诉应用程序去完成什么任务，有可能是一个文本框中输入一个名字。于是，伴随每个字符的输入，该应用程序将不断地接收到一个个事件信号，进而持续更新显示结果以反映所输入的名字。

对于窗体中定义的许多控件而言，应用程序将能够指定所采取的相关动作，这样操作系统就可以在没有应用程序介入的情况下完成大量的工作。例如，操作系统知道如何对带有增量式箭头的控件里的数值进行自动地递增操作。对于其他的按钮而言，应用程序则可能需要完成其他相应特定的处理。就应用程序定义的每个控件来讲，当控件被触及时，可能导致相应的事件编码传递给应用程序。具体以图 4-7 所示的确认式对话框的例子来说，当该控件被显示出来且用户触及其中一个按钮时，一个事件将会被发送给应用程序，且发给应用程序的数值将会标示出对应发出事件的控件以及被点击的按钮。

由于触摸屏的操作与应用程序之间是异步的（也就是说，在程序运行的同时，随时都会发生触屏事件），而且不少事件的发生比应用程序对它们的处理要快，所以操作系统必须要维护一个关于已经发生但尚未交给应用程序的事件的队列。这个队列按优先级次序进行维护，从而使相对重要的事件能够优先得到应用程序的处理。

一些此类的事件属于系统相关的事件，例如，当关闭电源（也就是说，系统即将进入低

功耗睡眠模式)时发送给应用程序的事件。这种情况下,应用程序将会暂缓所有其他操作,如与其他系统的通信等。

85

4.9 小结

本章中,我们讨论了一种简单的现代操作系统——由Palm公司开发的Palm操作系统™——的特征和概念。这种操作系统是针对小型手持设备而开发的。尽管这是一种单用户系统,但其可以并发执行若干操作系统进程和少量的应用程序。为此,它支持有限的并发执行任务,从而使其成为一种简单的多任务系统。

我们以Palm操作系统的概要介绍来开启本章的内容,然后讨论了使用Palm操作系统的手持式计算机的一些独特的硬件特性。这些与众不同的特性促成了Palm操作系统设计中所做的一些决定及选择方案。在此基础上,我们讨论了多任务的本质,还有操作系统是如何运作和调度应用程序进程及操作系统任务的。紧接着,我们讨论了内存管理以及操作系统所支持的不同类型的内存存储器。由于Palm平台通常并没有硬盘,而是使用所谓存储型随机访问存储器的内存的一部分来保存持久性数据。当电源关闭时,系统进入睡眠模式,存储型随机访问存储器依然保持其现有内容。我们讨论了内存是如何划分为内存块的,操作系统是如何在内存表中定位不同的内存块以及如何使用紧凑技术来管理空闲内存的。

随后,就Palm操作系统中的文件的组织以及Palm操作系统所提供的基本输入/输出功能的范畴进行了概要描述。后者涵盖允许用户以手写方式输入文本的涂鸦式输入系统。我们还描述了显示子系统和简单的图形化用户界面程序设计,扼要讨论了面向事件的程序设计——大多数Palm应用程序所采用的程序设计范式,另外还介绍了Palm操作系统中典型的应用程序的设计要领。

下一章,我们将继续介绍一种比Palm操作系统更复杂的操作系统。通常情况下,该操作系统在应用程序层级并发运行多个程序。为此,该操作系统相对来说要更为复杂些,且包含更多的系统开销。

参考文献

- | | |
|---|---|
| AMX/FS File System User's Guide. Vancouver, BC, Canada: KADAK Products Ltd., 1995. | Palm OS Programmer's Companion, Volume 1, Document Number 3120-002. Sunnyvale, CA: Palm, Inc., 2001. |
| AMX User's Guide. Vancouver, BC, Canada: KADAK Products Ltd., 1996. | Palm OS Programmer's Companion, Volume 2, Communications, Document Number 3005-002. Sunnyvale, CA: Palm, Inc., 2001. |
| Exploring Palm OS: Palm OS File Formats, Document Number 3120-002. Sunnyvale, CA: PalmSource, Inc., 2004. | Rhodes, N., & J. McKeehan, <i>Palm Programming: The Developer's Guide</i> , 1st ed., Sebastopol, CA: O'Reilly & Associates, Inc., 2000. |
| Exploring Palm OS: System Management, Document Number 3110-002. Sunnyvale, CA: PalmSource, Inc., 2004. | SONY Clie, Personal Entertainment Organizer. Sony Corporation, 2001. |
| Palm OS® Programmer's API Reference, Document Number 3003-004. Sunnyvale, CA: Palm, Inc., 2001. | |

网上资源

- | | |
|---|---|
| http://www.accessdevnet.com (Linux平台开发套件) | http://www.pocketgear.com/en_US/html/index.jsp |
| http://www.freewarepalm.com (免费的Palm软件) | (针对移动设备的软件) |
| http://oasis.palm.com/dev/palmos40-docs/memory%20architecture.html | http://en.wikipedia.org/wiki/Graffiti_2 (关于Graffiti 2 |
| http://www.palm.com (Palm公司主页) | (第2版涂鸦式输入工具软件)的文章) |

86

习题

- 4.1 既然 Palm 处理器任何时候在显示器上只能有一道程序，为什么该系统还需要多道处理操作系统？
- 4.2 相对于现代系统而言，除了处理器速度缓慢、内存容量非常小以外，什么是 Palm 操作系统所基于的主要硬件设计的最为独特的部分？
- 4.3 Palm 操作系统是微内核还是整体式内核？
- 4.4 为什么 Palm 操作系统要采用实时内核？
- 4.5 大多数 Palm 应用程序的基本逻辑流程是怎样的？
- 4.6 为什么分配给进程的内存是间接地通过主控指针表来进行访问的？
- 4.7 Palm 操作系统是如何对空闲内存进行监测的？
- 4.8 正如在大量的信息系统技术中常见到的那样，Palm 操作系统的开发者们也对一些非常好的术语进行重用并赋予其他不同的含义。Palm 操作系统文档在提到“数据库”（database）时，具体指的是什么意思？
- 4.9 既然 Palm 系统平台并没有多大的内存空间，为什么通常情况下 Palm 系统并不对数据库采取压缩措施呢？
- 4.10 Palm 操作系统为程序设计人员提供了关于输入 / 输出的若干种抽象机制，从而使应用程序员无须顾虑硬件细节。本章都提到了哪些抽象机制？
- 4.11 屏幕是基于内存来进行映射的而非通过输入 / 输出指令来进行处理的，为此，大多数应用程序会直接把数据移送到内存中的屏幕区域。这是否正确呢？
- 4.12 简要描述面向事件驱动的程序设计。
- 4.13 应用程序员是如何绘制其要在屏幕上显示的窗体的？

单用户多任务 / 多线程操作系统

5.1 引言

Mac 操作系统是我们的操作系统螺旋式演化案例的一个典范，因为它本身就经历了一系列的演变和发展。其一开始的目标是成为一个人人们可负担得起的拥有图形化用户界面的个人计算机的操作系统。在当时，这是颇具创新和革命性的。当然也有其他系统在使用图形化用户界面，但相关系统一般都极其昂贵。而且，除了图形化用户界面之外，在大多数方面，Mac 操作系统的第一个版本都不如第 4 章所讨论的 Palm 操作系统那么复杂。尽管如此，随着时间的推移，来自其他系统的压力导致了 Mac 硬件和 Mac 操作系统的演变，在 Mac 操作系统发展轨迹的最后，其功能大致与我们将在下一章所讨论的多用户 Linux 操作系统一样强大。二者的不同之处在于，Linux 操作系统从一开始就设计成支持多个并发用户，并且由此在相应结构和设计方面造成二者之间的一些显著差异。因此，我们把 Mac 操作系统作为从 Palm 操作系统（面向资源稀缺的平台环境，只有单个用户，多任务但没有用户多线程，有限的屏幕大小，没有辅助存储器）过渡到 Linux 操作系统（面向多用户、多任务、多线程，相关环境拥有大容量的辅助存储器和第三级存储器，图形化用户界面支持大屏幕和重叠窗口）的中间步骤来展开讨论。

89

鉴于 Mac 操作系统在其发展过程中曾经历过几次深刻的变革，所以我们在本章将使用不同于其他的螺旋式教学章节的方法来对其进行介绍。

在这一章中，我们从第 5.1 节起，首先概要论述 Mac 操作系统及其原始内核的基础结构相关的一些背景知识。简短介绍之后，将会在第 5.3 节一直到第 5.12 节的各节中按照 Mac 操作系统的版本演化依次阐述各个版本的新增功能和特征。这主要是考虑到，Mac 操作系统从一开始的简陋版本——大多数功能也就比 CP/M 好那么一点——起家，最终却演变成一个功能齐全、能够支持多个用户和多个进程且与下一章所讨论的 Linux 操作系统完全趋同的现代操作系统。追随 Mac 操作系统的发展演化过程本身就是一个微螺旋方法。其间，我们将在 X 版 Mac 操作系统处暂告收尾，并将在下一章转向阐明另一种系统，即 Linux 操作系统。关于 X 版 Mac 操作系统，我们将着眼于让其和当前市场上的其他主要个人计算机操作系统联系起来的目，展开充分但简洁的介绍。最后，我们在第 5.13 节对全章内容进行归纳总结。

5.2 Mac 计算机的起源

1973 年，一款名为阿尔托（ALTO）的创新性计算机系统在施乐帕洛阿尔托研究中心（Xerox Palo Alto Research Center, Xerox PARC）设计成功。这种计算机从未出售过，但是有 200 多台赠送给了大学和其他的研究机构。这些计算机以大约每台 32 000 美元^①的价格构建起来，其中包含有诸如我们今天所知道的图形化用户界面的先导原型等创新性技术、一种

① 相当于 2007 年的 157 000 美元。

以太网和一个鼠标，等等。而后来的一款系统，即 Xerox Star（施乐之星），也包含许多相同的功能，不过其零售价已经降到了 16 600 美元^①。但是，这对于一台仅仅设想由一个人来使用的计算机来说，仍然太过昂贵，故而有关系统并没有取得商业化的成功。尽管如此，这些系统得到了个人计算机业务中一些有远见卓识的先驱的关注，他们开始生产个人可负担得起且融入前述理念的系统。史蒂夫·乔布斯（Steven Jobs）就是其中的一位先驱者，他的苹果电脑系统位列第一批商业化成功的个人计算机当中。

苹果公司首先开发出了 Lisa（丽莎）计算机系统，其零售价为 10 000 美元^②。就像施乐之星一样，这也属于一件商业化失败的案例。然而苹果公司坚持不懈，并最终在 1984 年推出了 Mac 个人计算机，其零售价为 2500 美元^③，与 IBM PC 价格相当。对于普通人来说，Mac 比 Lisa 更实惠，而且有关图形化用户界面使其成为一个非常有用的系统，所以其取得了立竿见影的成功。需要说明的是，Mac 硬件采用了 Motorola 68000 系列处理器。

90

5.3 Mac 操作系统——第 1 版系统

Mac 操作系统的初始版本称为 System 1（即第 1 版系统）。第 1 版系统拥有若干项当时的操作系统的典型功能和特征。同时，由于其图形化用户界面，所以还有一些独特的特点。

5.3.1 图形化用户界面

第 1 版系统提供了一个桌面、一个鼠标以及若干窗口、图标、菜单和滚动条，如图 5-1 所示。在桌面上有一个垃圾箱图标，可用来删除东西，具体操作是拖动相应东西并投放到该图标上即可完成删除功能。这些都是现在我们认为理所当然的隐喻和功能，然而在当时却是颇具创新性的突破。与 Palm 操作系统不同，第 1 版系统设计假定有关屏幕足够容纳一个以上的窗口，或者能够显示桌面以及一个未占满整个屏幕的窗口。屏幕只有黑色和白色，且只有 520×342 像素的分辨率，所以图形功能非常有限。然而无论如何，这毕竟是一个图形化用户界面，并且许多用户特别是初学者发现它比命令行界面更加友好。在这里，不妨将其与 CP/M 中的命令行提示比较一下，后者只会显示出：

A>

然后就等待用户输入，而且没有给出任何提示来说明要做什么。

图形化用户界面可能是关于 Mac 操作系统最值得注意的一个方面，这不是因为它的想法如何新颖或者它做得如何出类拔萃，而是在于那些不必非得支持的东西。进一步说，在全世界的其他地方，操作系统通常情况下都是从原来具有命令行界面的传统系统（譬如 DEC、UNIX、IBM 等）演变而来，且相关应用程序都是通过称为命令行的接口上输入一行命令而调用的独立程序。这些接口模拟了连接到计算机的打字机的工作方式，故此它们都是围绕键盘使用而设计的，很少提供甚至没有鼠标支持。每个应用程序团队都可以针对特定功能自由地使用任何他们所期望的按键。于是，为了调用拼写检查功能，一个字处理程序可能使用了 F7 按键，而另一个电子表格程序则可能使用了 F12 键。更为糟糕的是，在大多数应用领域并没有占据支配地位的软件包，因此，WordPerfect 文字处理程序可能使用了这个按键来

① 相当于 2007 年的 42 000 美元。

② 几乎相当于 2007 年的 21 000 美元。

③ 相当于 2007 年的 5000 多美元。

启动打印功能，而像 WordStar 之类的竞争程序则可能会对同一项功能使用另一不同的按键。对于每个单独的应用程序而言，往往会提供键盘模板，以标明每个功能键在单独使用或者与 Shift、CTRL 和 ALT 键进行任意组合使用的时候，用来完成什么样的功能。

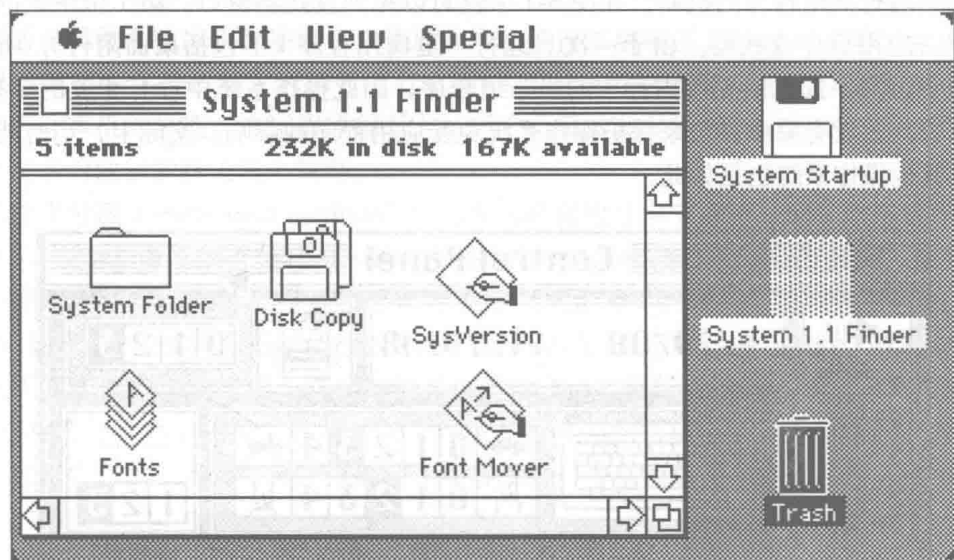


图 5-1 Mac 操作系统图形化用户界面

来源：本章中所有关于 Mac 操作系统的屏幕截图都是采用 Mini vMac 模拟器（针对早期 Mac 操作系统的模拟器）制作的。该模拟器可以登录如下网址获取：<http://minivmac.sourceforge.net/>

对于 Mac 操作系统而言，不存在任何前面遗留下来的应用程序。从一开始，其就为打印功能指定了一个按键序列，而一个新的应用程序没有理由背离相应的指定方案。于是，苹果公司可以实事求是地宣传在 Mac 操作系统上学习使用软件的方便。例如，假设一个用户已经在 Mac 系统上掌握了字处理应用程序，而且该用户理解电子表格工具的工作机理，那么该用户将能够很轻松地在 Mac 系统上使用电子表格程序，因为所有标准功能都将会以与字处理程序同样的方式进行调用。即使在今天，这个问题仍然存在于 Windows 和 Linux 应用程序中。关键在于，人们不应当低估有关向下兼容（backward compatibility，或称为向后兼容）要求的影响——而这是 Mac 操作系统所没有的。

5.3.2 单任务

为了交付普通民众能够买得起的产品，早期的 Mac 系统不得不在非常有限的内存条件下运行，毕竟它仍然十分昂贵。鉴于此，苹果公司的开发人员决定摒弃曾经在 Lisa 计算机上所使用的多任务功能。进一步说，即便一个应用程序的窗口没有占满整个屏幕，Mac 操作系统最初也不允许多个程序同时运行，甚至对于后台的打印功能也是如此。为了使用某些并行的功能，操作系统包含了若干桌面附件，其中包括计算器、闹钟、系统控制面板以及记事本，不过它们都进行了限制，以确保不会占用太多的内存。准确地说，它们是以“设备驱动程序”方式而不是以单独的程序实现的，并且可以打开一个窗口。图 5-2 显示了以前的计算

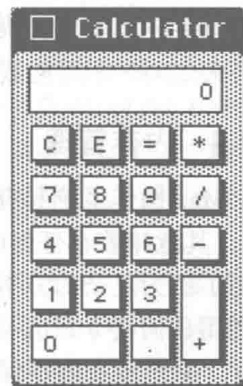


图 5-2 “计算器”桌面附件

器样式，不言而喻，以今天的标准来看确实是有点原始。图 5-3 则给出了以前的控制面板，其能够支持用户更改许多系统设置。有关系统中还有一个称为查找器的应用程序，用于在系统中查找文件。查找器窗口是命令处理器，在大多数操作系统中体现为一个命令行控制台，故而它也是运行其他程序的机制。在图 5-1 中就可以见到查找器窗口。第 1 版系统的查找器被称为单一应用程序查找器。由于一次只运行一道应用程序（不包括桌面附件），所以没有必要防止一道程序读取或更改内存中的另一道程序，因此操作系统中没有相关的保护设计。第 1 版系统甚至没有采取措施来保护操作系统免受应用程序的影响。实际上，当时能够使用的大多数其他操作系统也是如此。

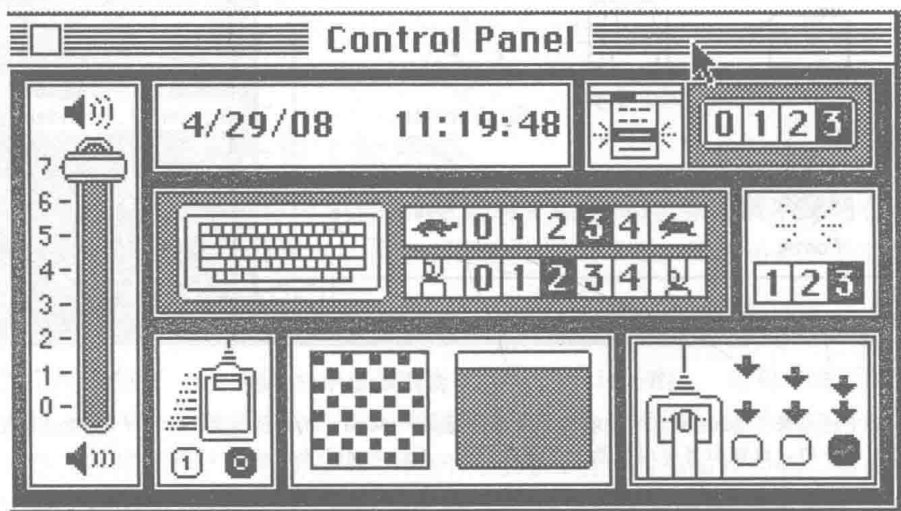


图 5-3 控制面板

5.3.3 辅助存储器

与第 3 章所讨论的 CP/M 系统一样，第 1 版系统的程序保存在单个软盘驱动器上，并且只有在执行的时候才会被加载到内存中。需要强调的是，早期 Mac 系统上可用的磁盘系统只有 400KB。这是一个足够小的空间，很容易就能够找到文件，故而所有的文件都被保存在一个单一的目录里。但是，操作系统的开发人员意识到把相同的文件分组到一起的设想非常有用，所以有关系统就显示了磁盘上的文件夹（folder）。尽管如此，与 CP/M 系统一样，这些文件夹仅仅是一种模仿。具体而言，每个文件目录项可能被标上了文件夹名称，并且系统允许用户查看文件夹里的内容，其实质就是列出标记有该文件夹名称的所有文件。为此，文件夹之间是不可能嵌套的。

5.3.4 内存管理

Mac 操作系统具有单一的地址空间，如图 5-4 所示^①。这种体系结构被认为是“平坦型的”，其意味着在任何时候任何指令都可以直接引用所有的内存。在那个时代的其他设计则采用了更为复杂的方案且允许有更多的内存（RAM），不过对寻址进行了限制，使得相关程序利用任何指令只能寻址 64KB 大小的内存范围。68000 处理器具有 24 位地址，支持 16MB 内存。系统没有内存保护，故而任何程序都可以修改内存中包括操作系统本身在内的任何东

① Mac 操作系统的最初版本并未支持多进程，后来才引入多进程。

西。同时，应用程序代码也运行在管理程序模式（supervisor mode，或称为管态）下，所以没有指令保护机制来对应用程序可以完成的事情加以限制。另外，在操作系统启动时，便确定了地址空间的大小。具体而言，内存的最低地址部分由系统分区（system partition）占用，这一区域包含有一些系统全局变量，且应用程序按理说应当不能够直接访问。换言之，应用程序应当通过操作系统应用程序接口来读取和操作任何系统数据。然而实际上，由于没有任何内存保护或指令保护机制，所以没有什么能够阻止应用程序采取捷径和直接访问相关信息。在个人计算机的早期，应用程序编写人员通常会采取此类快捷访问方式，并试图以提升性能的名义来为他们的行为做出解释。

应用程序分区（application partition）从内存的最高地址向下进行分配。应用程序分区的布局如图 5-5 所示。具体而言，其顶部是一个固定大小的数据块，称为 A5 定位区（A5world），包含有应用程序的静态数据和一些关于应用程序的元数据。之所以如此命名，主要是考虑到 Mac 操作系统会利用指向该区域的指针来对处理器的 A5 寄存器进行加载，这样应用程序就可以获悉其在内存中的位置，并且能够通过相对于 A5 寄存器的寻址来访问其全局数据。在 A5 定位区之下是栈，栈的“顶部”向下扩展。同时，堆则从应用程序分区的底部向上增长，且包含有代码段。因此，操作系统必须要解决的一个问题就是确保这两个区域不会发生重叠。

94

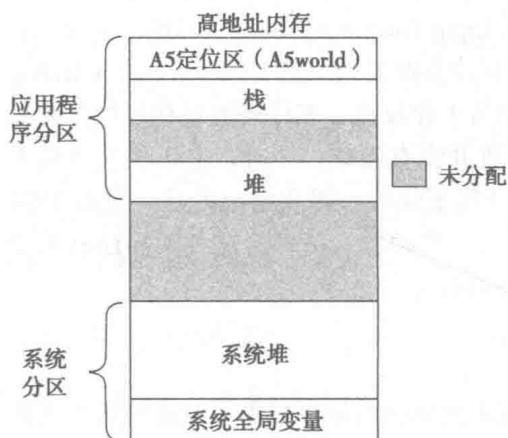


图 5-4 第 1 版系统内存布局

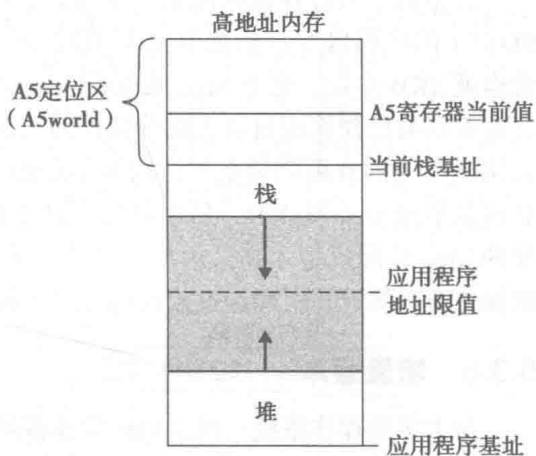


图 5-5 应用程序内存分区

在应用程序启动时，一般会为堆的大小设置一个上限。而堆的增长由内存分配例程加以控制，所以相关例程便不断地检查以确保请求分配不会超过该上限。然而栈由硬件来自动维护。伴随子例程以及函数的调用和返回，数据被压入栈里和从栈中弹出。鉴于许多应用程序会调用多个级别的子例程，有时还是递归式调用，所以这种栈将会随着程序的运行而不断增长。但是，没有硬件保护机制能够防止栈的扩展超过堆的上限。为此，系统设立了一个栈嗅探器（stack sniffer）子系统，在监视器垂直回扫（大约每秒 60 次）的间隔期间运行和检查栈增长水平是否触碰到了堆的上限。

如何最佳使用 128KB 的内存是 Mac 系统设计人员所面临的一个大问题。从某种意义上讲，这是一笔大量的内存。同一时代的其他个人计算机也就具有 16KB 或 64KB 的内存。但是，Mac 操作系统的目标是要拥有图形化用户界面，而此类界面往往会耗用不少的内存。正如上面所提到的，开发人员决定限制 Mac 系统一次只运行一个程序。于是，他们最关心

的似乎是内存碎片化——不断的内存分配和回收常常会导致许多小的被隔离的内存区域，由于它们各自太小，所以即使加起来总的空闲内存可能足以满足特定请求也不能得到利用。为了避免堆内存的碎片化，Mac 操作系统支持可重定位的内存块。这些内存块通过指向不可重定位的**主指针块**（master pointer block）的指针来间接访问。上一章讨论的 Palm 操作系统也使用了类似的机制。在内存回收过程中，可重定位块经常会被紧凑处理。可重定位块也可以标记为可清除的，从而意味着如果空闲内存空间缩减到低于预定的限值，那么系统就可以在紧凑期间把它们释放掉。指针最初只有 24 位长，但是考虑到处理器在可预见将来的字长扩展态势，故而将其存放在了 32 位字段中。进一步说，（32 位中的）高 8 位通常用作标志位以将对应块标记为可重定位的、临时的、可清除的，等等。

95 操作系统采用上述方案实现了**系统堆**（system heap）以及**应用程序堆**（application heap）两个区域。只要仅有一个程序运行，系统一般能够工作得很好。同时，鉴于应用程序堆在程序退出时进行了清除，所以碎片化问题被降低到了最小程度。然而不幸的是，正如我们上面一再提到的，操作系统没有提供任何内存保护措施，所以由应用程序错误操纵系统堆而导致的崩溃情况并不少见。

5.3.5 只读存储器

大多数个人计算机仅使用少量的只读存储器来存放用于开机自检（Power-On Self-Test, POST）的代码以及一些基本输入/输出系统（Basic Input/Output System, BIOS）例程，通常也就 8KB 左右。对于 Mac 操作系统而言，相应的只读存储器则明显要大些（大约 64KB），并且保持有操作系统自身大部分的代码。在只读存储器中存放这么多代码的最初目的是避免占用软盘上的有限存储空间，因为早期的 Mac 计算机并没有硬盘。同时，鉴于相关代码不必再从软盘驱动器读取，所以这也有助于使系统启动得更快些。需要指出的是，只有 1991 年的 Mac 标配机型才是采用单独只读存储器的方式进行启动的。另外，这种架构还有助于确保只有苹果计算机和经过授权的克隆才能运行 Mac 操作系统。

5.3.6 增量版本

与大多数操作系统一样，Mac 操作系统在主要版本之间也有增量版本，且这些版本通常标以小数形式。进一步说，相关版本主要是出于如下各种原因而发布的：针对诸如操作系统加载等某项特定功能进行提速、错误修复以及为稍后某主要版本安排的某项新功能或应用程序遭遇有关主要版本发布延后的偶然情况。在 Mac 操作系统的第 1 版系统中就有一个这样的版本，即 1.1 版，相关发布缘由涉及了上述各个方面。

5.4 第 2 版系统

从理论上讲，第 2 版系统是一个主要版本，但其却没有任何从理论角度来看具有重大意义的特征。尽管如此，查找器确实要更快一些。同时还剔除了某些图标/命令，并新增了用于创建新文件夹和关闭系统的图标。软盘现在的弹出操作也更加便捷，仅仅需要将它们的图标拖动到垃圾箱（trash，或称为回收站）即可完成，而无须首先选择弹出磁盘命令然后再将相应图标拖动到垃圾箱。另外，还添加了一个选择打印机的桌面附件，以允许用户选择一台默认的打印机。该实用程序后来演化成为**选择器**，即用于访问共享资源（如打印机、调制解调器以及托管在其他系统上并通过网络提供的磁盘卷）的实用程序。

5.4.1 图形化用户界面

Mac 系统的用户喜欢图形化用户界面以及从一个应用程序剪切信息并粘贴到另一个应用程序的能力。不过，这意味着先得从一个程序中剪切数据，接着停止该程序，然后再启动新程序，并将相应数据粘贴到新程序当中——整个操作过程通常需要几分钟。每款新的 Mac 机型拥有比以往机型更多的内存，Mac 512K（Macintosh 512K，又名 Fat Mac）机型所包含的内存则达到先前 Mac 机型内存的 4 倍。这足以支持某种形式的多任务，并率先在切换器程序中得到了实现。进一步说，切换器允许用户启动若干程序，然后，用户便可以通过点击菜单栏上的图标在这些应用程序之间进行切换，于是，当前应用程序将会沿水平方向滑出视图，而下一道应用程序紧接着滑入。当用户切换到正在运行的各道程序中的某一道程序时，该程序被称为持有焦点（have the focus）。在这种情况下，用户可以在几秒而不是几分钟实现在应用程序之间的剪切和粘贴操作。

96

5.4.2 多任务

切换器在内存中创建了若干可把应用程序加载到其中的固定插槽。对于用户启动的每道应用程序而言，切换器程序将会为之分配一个单独的堆，显然，用户启动的应用程序数量会受到可用内存空间的制约。当用户从一个进程切换到另一个进程时，切换器可以执行上下文切换并准备操作系统内存管理数据，以便使操作系统开始运行新的应用程序。由于没有内存保护或指令保护，所以切换器可以调整操作系统内存结构来实现切换。尽管如此，这是非常受限的多任务，有点像 Palm 操作系统，因为在任何时间仍然只有一个进程运行。是的，用户可以从一个进程切换到另一个进程，但是当进程没有持有焦点时，有关进程实际上并没有运行。不过话又说回来，尽管这种方案很笨拙，但其与当时系统的内存管理方案相处得还行，即程序无须进行改变就能与切换器协同运行。同时，相关更改对操作系统内核也是透明的。在第 2 版系统中具有多个进程的典型内存布局如图 5-6 所示。

97

5.5 第 3 版系统

5.5.1 多级文件系统

磁盘驱动器存储空间变得越来越大，而用户当时常常倾向于把磁盘填满，就像用户现在所做的一样，同时还希望能够快速地访问他们的所有信息。需要强调的是，这意味着文件数量急剧地增长，因而让用户变得难以跟踪相关文件。在这种情况下，一种新的文件系统设计方案得以发布，称为多级文件系统（Hierarchical File System，HFS，或称为分层式文件系统）。它替代了旧的 Mac 文件系统（Macintosh File System，MFS）。其间，文件夹成为实实在在的子目录，而不再仅仅是目录项中的标签，并且文件夹能够包含其他的文件夹。这种文件系统更加令人满意，并开始被称为 Mac 操作系统标准文件系统以区别于后来的扩

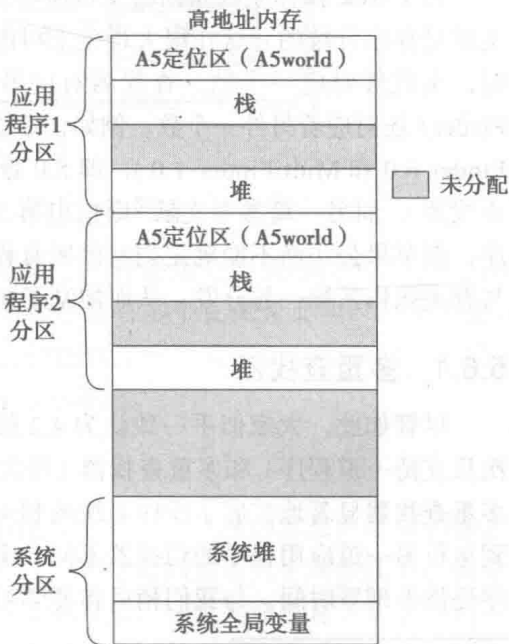


图 5-6 第 2 版系统“切换器”内存布局

展版本。进一步说，有关目录项还包含有标明文件何时创建和何时最后一次修改的时间戳、文件类型和创建程序代码、最多 32 字符的文件名以及其他的文件元数据。（创建程序代码会告知操作系统是什么应用程序创建了相应文件。）另外，有关系统采用位图来监测空闲空间，并运用 B 树来存放目录。相关思想将在第 12 章和第 13 章展开进一步阐述。

期间，曾发布过几个错误修复版本（或称为补丁版本），直到有关操作系统功能又一次真正提高。

5.5.2 网络

局域网（Local Area Network, LAN）逐渐流行开来。一方面，局域网允许共享访问诸如大型磁盘驱动器、高端激光打印机、调制解调器池等价格昂贵的设备以及譬如微缩胶片输出等其他异乎寻常的设备。另一方面，局域网还通过中央服务器上的共享文件及目录有效地促进和方便了通信。因此，在 3.3 版系统中，苹果公司添加了对苹果共享机制（AppleShare）——一种专有的文件共享协议——的支持。有关协议栈还包括其他层级的专利技术，即网络层的苹果对话（AppleTalk）协议以及数据链路层和物理层的本地对话（LocalTalk）协议。现在的选择器实用例程也比当初只是用来选择默认打印机而变得更为重要。进一步说，LaserWriter（译者注：苹果公司激光打印机品牌名）打印机可以直接连接到网络上并被多个用户共享。Mac 开始被视为一个强大的桌面排版系统，而相关打印机则是这种看法及 Mac 产品线全面成功的一个重要因素。

5.6 第 4 版系统

第 4 版系统首先安装在了 Macintosh SE 和 Macintosh II 计算机上。在当时的操作系统技术开发阶段，新的操作系统版本通常只要求支持新型计算机。再比方说，4.1 版系统就主要增加了对大于 32MB 的磁盘驱动器的支持。

关于 Mac 操作系统是何时支持能够启动多个应用程序的查找器版本的问题，不同的参考文献是存在分歧的。这在很大程度上可能是因为相关版本的命名有些混乱。具体而言，在当时，主软件对应一个数，查找器对应另一个数，而稍后马上要讨论的多重查找器（Multi-Finder）还对应着另外一个数。例如，一篇参考文献^①列出了 System Software 5.0（System 4.2，Finder 6.0 和 MultiFinder 1.0），即 5.0 版系统软件（4.2 版系统，6.0 版查找器和 1.0 版多重查找器），而另一篇参考文献^②则指出第 5 版系统从未发布过。此外，由于多重查找器是新程序，而苹果公司尚不能确定当时的所有程序可否在其下面正确运行，所以原先的查找器继续与有关操作系统一起分发，从而加剧了相关版本命名问题的严重性。

5.6.1 多重查找器

尽管如此，大家似乎一致认为 4.2 版系统实现了多重查找器——用户可以在查找器（每次只支持一道程序）和多重查找器（每次可以支持多道程序）之间进行切换，如图 5-7 所示。多重查找器显著地扩展了操作系统的相关功能。与仅仅将操作系统从运行一道应用程序切换到运行另一道应用程序的切换器不同，多重查找器允许每道程序保持运行，并为每道应用程序提供处理器时间。与我们稍后将要学习的操作系统也不相同，Mac 操作系统没有对一个进

① http://en.wikipedia.org/wiki/Mac_OS_history

② http://www.macos.utah.edu/documentation/operating_systems/mac_os_x.html

程能够继续运行多长时间而不切换到另一个进程设置硬性限制。在 Mac 操作系统中所采用的技术称为协作式多任务 (cooperative multitasking)。利用这种技术, 一个进程可以想运行多长时间就运行多长时间。如果有关进程调用了操作系统, 而操作系统不能立即提供相应服务, 譬如磁盘读取, 那么操作系统将会让有关进程等待——一种称为阻塞 (blocking) 的机制。当进程执行了此类阻塞性调用时, 操作系统将会把被阻塞的进程添加到正在等待某项事件的进程队列中, 然后进行切换和运行另一个进程。而如果一个进程没有执行任何阻塞性调用, 那么对应进程可以想运行多长时间就运行多长时间。为了让所有进程都能够对用户的请求做出快速的响应, 所有进程都需要一些处理器时间。换句话说, 如果一个进程运行时间太长, 就有可能使系统性能失去均衡和不再稳定。为了防止这种情况的发生, 所有进程都应该经常执行一条特殊的系统调用, 以告知操作系统尽管相应进程本身尚未结束, 但是将主动放弃控制并准备好了再次运行。这便允许其他的进程拥有公平合理的处理器时间份额。当然, 一些供应商往往希望自己的软件看起来是响应最好的, 故而并不经常调用有关机制。另外在某些情况下, 软件错误可能会导致程序陷入循环, 故而使有关程序永不放弃控制或从不执行阻塞性调用。一旦这些情况发生, 有关系统将基本上停滞不前。

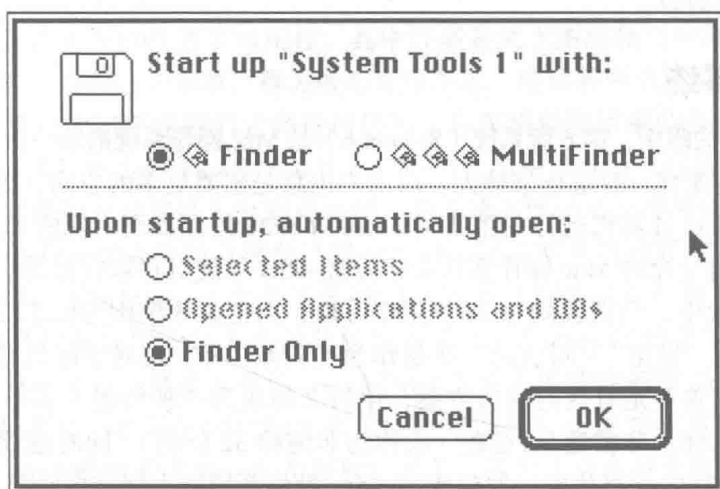


图 5-7 多重查找器

5.6.2 多重查找器与图形化用户界面

多重查找器为来自不同应用程序的诸多窗口提供了一种使用分层模型同时共存的方法。既然可能有多个运行的应用程序, 而它们可能每道程序都有多个在桌面上同时打开的窗口, 那么, 当一道程序得到焦点时, 它的所有窗口将会被往前提到一起进而分布在同一个层上。这是为了与已有视窗机制的应用程序接口保持兼容而必需的。

5.6.3 内存管理与多重查找器

多重查找器还为应用程序提供了一种向操作系统传达其内存要求的方法, 以便多重查找器可以根据每道程序的需要为其分配内存。可惜的是, 被指定的数量对于某些任务来说并不够用, 于是便向用户提供了一个否决和重置有关数量的界面。这大大地违背了苹果公司关于用户应当远离此类技术信息的相关理论。就此案例而言, 他们的理论无疑是正确的, 因为用户在一般情况下并不清楚一道程序实际可能需要多少内存。为此, 经常是分配给这道程序比

其实际需求还要多的内存，而分配给另一道程序的内存量则又太少。相应结果是，“饥饿的”应用程序运行表现非常不好。客观而言，在多道应用程序同时运行的情况下，内存管理通常比单道应用程序运行的情况要复杂得多。但是在开发多重查找器的时候，一个关键的考虑因素是在单个的查找器下正常运行的程序应该不经改变就可以在多重查找器下正常运行。因此，二者的内存架构非常相似，只是后者要稍微复杂一点而已。关于运行一道应用程序的内存体系结构如图 5-4 所示。而当多道应用程序同时运行时，相应的内存体系结构如图 5-6 所示。当执行从一道应用程序切换到另一道应用程序时，操作系统将会改变某些系统变量的内容以反映新应用程序的应用程序分区及各组成部分的大小和位置。这种更改称为上下文切换 (context switch)。正如我们后面将会看到的那样，现代操作系统的上下文切换通常比这要更为复杂。

5.7 第 5 版系统

如上所述，一些参考文献宣称，第 5 版系统从未发布过，而其他的一些文献则认为其发布仅限于很短的一段时间。无论哪一种情况，从研究和学习操作系统演化的角度而言，都没有什么值得关注的内容。

5.8 第 6 版系统

在许多观察者的眼中，第 6 版系统 (System 6) 是 Mac 操作系统的第一次真正升级^①。随机存取存储器越来越便宜，容量越来越大，而用户也总是想要更多的空间。因此，第 6 版系统开始迁移和支持 Mac 计算机运行在伴随 Motorola 68020 处理器而出现的真正的 32 位内存寻址模式下。32 位地址允许 Mac 操作系统最多寻址 4GB 的随机存取存储器。然而，在早期版本的 Mac 操作系统中，只把较低的 24 位用于寻址，而把高 8 位作为标志位和用来说明所指向的块是被标记为“锁定”“可清除”还是作为“资源”等。这对于随机存取存储器十分有限的早期硬件而言曾经是有效的解决方案，但后来却成为一种负担。苹果公司把采用 24+8 位寻址模型的代码称为非彻底 32 位的 (或称为非纯粹 32 位的)，同时建议相关开发人员从他们的应用程序中移除此类代码。如前所述，Mac 操作系统的大部分代码都在只读存储器中。令人遗憾的是，那部分只读存储器代码大多数是非彻底 32 位的，故而老旧的 Mac 计算机不能迁移到这种新模式下。这种新模式需要新版本的硬件。与此同时，改变为 32 位寻址模式还产生了许多兼容性问题，甚至还继续存留在今天的 Mac 操作系统版本中。进一步说，操作系统继续保持着以 24 位模式运行应用程序的能力，虽然该模式要比 32 位模式慢很多。也正因如此，苹果公司这时候已开始感觉到支持传统应用程序的重压。

在个人计算机时代的早期，开发人员仍然将系统中的随机存取存储器看作非常紧张的资源，并且会竭尽全力地去节省一两个字节。但随着时间的推移，经常发现这样的节省后来往往会造成一些非常负面的影响。事实上，千年虫 (Y2K bug 或 Year 2000)^②就是由于希望节省几个字节的随机存取存储器而通过将日期的年份部分的格式缩短成最后两个字节所引起的此类问题的另一个例子。当时，尽管距离世纪末还有 20 年左右的时间，但开发人员普遍认为他们开发的系统到那时候将不再正常运行。进一步说，当世纪末最后一年即将结束的时候

① http://en.wikipedia.org/wiki/Mac_OS_history

② <http://en.wikipedia.org/wiki/Y2k>

候, 日期年份存储为两位数的系统将会得出不正确的结论, 即认定“00”(即2000)年的日期反而在“99”(即1999)年的日期之前。显然, Mac 操作系统从一开始的设计就避免了千年虫问题, 尽管苹果公司从未正式证实第7版系统(System 7)之前的任何系统版本是正确满足2000年年份存储要求的。

5.9 第7版系统

第7版系统是到那个时候为止针对有关系统软件所做的最大改变。它继续实现 Mac 操作系统到彻底的32位寻址方式的迁移, 改进了对彩色图形、网络和多任务(multitasking)的处理, 并引入了某种形式的虚拟内存(virtual memory, 或称为虚拟存储器)。

与此同时, 在早期版本的 Mac 操作系统中提供作为选项的许多功能被集成到了第7版系统中。该版系统还抛弃了查找器的单一程序版本, 从而消除了查找器与多重查找器对决的问题。于是, 协作式多任务成为系统的正常运作模式。基于 AppleTalk 的网络支持以及基于 AppleShare 的文件共享也不再是作为选项, 而是被嵌入了操作系统内部。

[101]

5.9.1 图形化用户界面

第7版系统在若干方面改进了可用性, 其中许多是关于图形化用户界面的。具体而言, 在菜单栏的右端新添加了一个菜单, 称为应用程序菜单。应用程序菜单给出了一个正在运行的程序的列表, 并允许用户在它们之间进行切换。在应用程序菜单的旁边是帮助菜单。同时, 用户现在可以执行更为简单的拖放操作——一个文本块可以用鼠标从一个程序拖到另一个程序, 而不必非得像以往那样先复制再粘贴。第7版系统的查找器最终利用了颜色特性, 从而使某些界面元素看起来更加立体化。除此之外, 第7版系统还添加了一些可用性功能。进一步说, WorldScript 为除了英语之外的语言提供了系统级的支持, 而同步发布的各种技术则具体包括用于任务自动化的宏语言 AppleScript (即苹果脚本语言)、颜色管理实用程序 ColorSync、多媒体软件 QuickTime 以及字体管理程序 TrueType (即字型标准)。随着时间的推移, 我们与现代的图形化用户界面进行交互的许多功能特征都被陆续地添加到了 Mac 操作系统中。在大多数情况下, 我们不会在每个版本中详细阐述相关功能。我们通常只会提示图形化用户界面正在不断演化发展, 并且随着时间的推移而变得更好用。

5.9.2 虚拟内存

有时用户希望同时运行比内存可容纳的程序数量还要多的程序。或者有关程序可能用到了一个非常大的数据文件。例如, 对于一个字处理程序来说, 如果其只是被用来编写办公室的备忘录, 那么通常很小的空间便可以存放得下; 但如果其用来编辑一份大型的报告, 那么它就可能需要大量的内存。进一步说, 如果需要较多内存但却由于内存过于昂贵等原因而没有较大内存可用的情况下, 系统性能往往会变得很差, 于是这时候可以采用一种称为虚拟内存(Virtual Memory, VM)的解决方案。虚拟内存是使用硬盘驱动器上的一些空间来模拟较大的主存的一种技术。它需要额外的内存管理硬件支持才能工作。简要说来, 内存被划分成所谓页面(page)的物理块。当程序开始运行的时候, 只有该程序的第一个页面被送入内存。而当正在运行的程序引用尚未进入内存的程序部分时, 相关硬件将会触发所谓缺页故障(page fault)的中断, 然后操作系统就会从磁盘驱动器把所缺的页面读到内存中。这种技术

将会在第 11 章展开更为详细的讨论。

刚才提到过，在计算机系统中需要特殊的硬件才可以使操作系统能够支持虚拟内存。具体而言，计算机必须具有特殊的内存管理单元（Memory Management Unit, MMU），其能够解释在处理器中运行的程序所生成的逻辑地址（logical address），并把它们转换为物理地址（physical address），故而程序的各个页面可以被安置在随机存取存储器的任何地方。苹果公司的基于 68040 处理器和 68030 处理器的机器在处理器中内置了支持虚拟内存的内存管理单元，因此在没有额外硬件的情况下就能够支持虚拟内存。Mac II 型计算机（基于 68020 处理器）可能在其主逻辑板（main logic board，简称主板）上具有一个特殊的内存管理单元协处理器，顶替了标准的地址管理单元（Address Management Unit, AMU）^①，该内存管理单元也支持虚拟内存。

102

对于 Mac 操作系统来说，虚拟内存是首先在第 7 版系统中实现的。然而，其虚拟内存支持非常初级，且在许多情况下执行得很差。进一步说，按照有关操作系统的内存管理器的设计方案，在虚拟内存机制下使用内存时，常常会导致发生过多的页面故障^②。同时，在现代的其他操作系统的虚拟内存实现中通常可以找到的虚拟内存功能——例如受保护的地址空间、内存映射文件、页面锁定、共享内存，等等——在第 7 版系统中尚未出现。换句话说，其中许多功能是在 Mac 操作系统的更高版本中才提供的。伴随苹果公司对虚拟内存工作原理的理解的加深及其对有关操作系统特定部分行为的修正，在运行虚拟内存时的系统性能也得到了持续的改善。

5.9.3 新型处理器

大致在 1990 年，苹果公司与 IBM 公司和摩托罗拉公司结成联盟，一起开发一种在结合了 IBM RS6000 架构、Motorola 68000 和英特尔个人计算机生产线的基础之上的新款处理器系列，其被称为 PowerPC（强力个人计算机）系列，并长期决定着苹果公司的硬件发展方向，直到 2006 年。最初采用 PowerPC 处理器的 Mac 计算机是 Power Macintosh 6100 或 Performa 6100 系列。而对这款处理器系列的支持则始于 7.1.2 版系统（System 7.1.2）。为支持有关处理器，Mac 操作系统的设计方案不得不进行必要的更改。具体而言，有关处理器架构是精简指令集计算机（Reduced Instruction Set Computer, RISC）设计方案，与摩托罗拉 68000 系列所采用的复杂指令集计算机（Complex Instruction Set Computer, CISC）设计方案不同，故而说明处理器所使用的代码已经发生了根本性的改变。考虑到要把基于 68000 架构的操作系统完全移植到精简指令集计算机架构上将会耗费特别长的时间，所以 PowerPC 架构的设计方案允许其自身对 68000 处理器进行仿真处理。

一小段被称为超微内核（nanokernel）的代码用来管理 PowerPC 处理器。其运行在管理程序模式下，并为系统的其余部分提供有关硬件管理的低级接口。该超微内核的应用程序接口仅限于系统软件和调试程序（或称为调试器）进行使用。在系统启动时，由超微内核启动 68000 仿真器，后者仅仅仿真 68000 用户模式指令集，而并没有对相关内存管理单元进行仿真，这样做有利于支持更普遍的兼容性。于是，有关操作系统几乎立即就能在基于 PowerPC 的系统上开始运行。当然，在 PowerPC 上仿真 68000 处理器的执行要远远慢于对 PowerPC 原生代码的执行。此外，相关程序可以被编译和链接生成既包含 68000 原生代码又包含

① <http://developer.apple.com/documentation/mac/Memory/Memory-152.html#HEADING152-0>

② <http://developer.apple.com/technotes/tn/tn1094.html>

PowerPC 原生代码的可执行模块。这便允许相应的单一版本的程序既可以在老旧机器上运行,又可以在新型机器上运行。这样的双模式程序被称为胖二进制代码(fat binary)。在这两种模式之间的切换是通过一组称为代码片段管理器(Code Fragment Manager, CFM)的库例程来完成的。久而久之,有关操作系统越来越多的代码被修改成为既包含 PowerPC 原生代码,又包含仍然可以在 68000 系列系统上运行的代码。

苹果计算机的体系结构一直是专有的,因而会产生几方面的效应,有些是正面的,但也有些是负面的。在 Mac 计算机中主要的苹果系统总线称为 NuBus(读作“New Bus”,即新型总线)。由于它是专有性质的,所以苹果公司可以对所有的硬件开发实施严格的控制。因此,相关控制器更可能在 Mac 计算机上而不是企业标准架构(Industry Standard Architecture, ISA)总线的机器上正常工作,同时相应的驱动程序也更可能在 Mac 计算机上正常运行。另一方面,这也意味着在这个市场上遇到的竞争较少,故而用户为有关硬件和软件支付的价格要高于他们可能为其他品牌计算机支付的价格。此外,只有极少的供应商能够承受得起去雇用额外的员工来开发针对其他总线的硬件。在 1990 年前后,英特尔开始着手开发一种称为外围部件互连(Peripheral Component Interconnect, PCI)总线的标准化总线。到 1993 年,相应完整的规范说明得以推出,插件厂商开始为这种新的总线构建输入/输出插件,系统制造商也开始将它们包含在新的主板设计当中。在这种情况下,苹果公司发现有关总线专有战略让他们在竞争中处于了劣势。具体来说,鉴于供应商可以在外围部件互连总线市场上销售的数量要明显大于在苹果新型总线市场上的销售数量,所以苹果必须得为接口控制器支付更高的价格,而这既减少了他们的硬件利润,又使得他们的系统价格降低了竞争力。为此,苹果计算机公司在 1995 年推出的强力 Mac 计算机中集成了外围部件互连总线。进一步说,7.5.2 版系统对这些新机器给予了支持,故而需为外围部件互连总线和控制器增加新的驱动程序和芯片集支持。

5.9.4 输入/输出增强

Mac 系统存在于一个由基于英特尔且运行着微软软件的个人计算机所主导的世界上。因此,Mac 系统面临相当大的压力去建立和提供通向相关计算机系统的“桥梁”。毋庸置疑,网络支持就是沿着相应方向不断演化和向前发展的,于是许多面向微软的协议被添加到了 Mac 操作系统的支持当中。另一个例子则是,7.1 版系统引入了能够访问 MS-DOS 格式软盘的称为 PC Exchange(个人计算机交换)的软件。早期版本的 Mac 操作系统只支持苹果版软盘格式。虽然 IBM PC 和苹果公司 Mac 计算机的软盘在物理上是相同的,但它们却以两种不同的方式进行使用。首先,二者的低级格式化(low-level formatting)不同。新软盘在大多数情况下并没有任何预先指定的扇区数或扇区大小。而一个被称为低级格式化的进程用来在磁道上写下扇区头部信息,从而供以后告知相关硬件每个扇区的开始位置以及对应扇区的磁道号、扇区号和大小。不同的系统可能使用不同数量和尺寸的扇区,并且早期关于介质的各种尺寸及低级格式存在着许多相互竞争的格式规定。现在,相关尺寸和格式已经相当标准化了,但在 20 世纪 90 年代初,则依然存在着若干相互竞争的标准。一旦完成了低级格式化,有关用户就可以使用操作系统在更高级别上对软盘进行“格式化”,即在磁盘上创建一个空的文件系统。对于 IBM 公司和苹果公司的系统来说,二者的文件系统也是不同的,这一点就像二者的低级格式化一样。鉴于用户之间方便轻松的文件交换非常必要,所以在 Mac 操作系统中添加了读写 MS-DOS 软盘的能力,从而使 Mac 系统在办公室领域更容易受到认

可和被接纳。

到了这个时候，笔记本电脑系统已经被频繁使用，而且它们往往会包含一个个人计算机卡（PC card，或称为 PC 卡）的插槽。这些插槽当时被称为 PCMCIA（Personal Computer Memory Card International Association，个人计算机存储卡国际协会）插槽，但后来被改名。个人计算机卡插槽允许插入一部未内置在原先的笔记本电脑中的设备。典型的例子包括网卡、外部磁盘驱动控制器、磁盘驱动器本身以及内存卡（RAM 卡，或称为随机存取存储器卡）。内存卡无法作为主存而被寻址，因为 PCMCIA 插槽位于输入/输出总线上。因此，一种处理这种卡的常见技术是将其视为一种特殊类型的磁盘驱动器并在其上面创建文件系统。考虑到软盘格式大小基本合适，所以这些内存卡通常采用 MS-DOS 兼容文件系统来进行创建，这样它们随后也可以用于把数据从 IBM 兼容的个人计算机上迁移到 Mac 计算机上，因为 Mac 操作系统也可以读取这些设备。

由于对有关操作系统常规功能的增强以及面向与强力个人计算机处理器协同使用的胖二进制代码，第 7 版系统成为 Mac 操作系统的第一个完整安装太大而无法在 1.44MB 软盘上容纳下的版本。其结果是，7.5 版系统及更高版本不再从软盘驱动器运行，而是要求相应计算机上拥有一个硬盘。

5.10 第 8 版系统

这时候，苹果公司正在把 Mac 系统添加到他们的旨在用作服务器的产品系列中。在某些情况下，这些新系统拥有多个处理器。因此，第 8 版系统增加了对这些新的 Mac 多处理器模型的支持。这些机器将会在承担服务器角色的过程中获得更好的性能。在现代操作系统中对此类系统的支持被称为对称多处理（Symmetric MultiProcessing, SMP）。在这种情况下，操作系统可以运行在任何可用的处理器上^①。这可能会给操作系统带来一些特殊的问题，因为有关操作系统可以真正地同时在两个或多个处理器上运行。换句话说，这意味着有关操作系统必须得采取特殊的预防措施，以防止有两个正在运行的实例同时操作任何一个数据元素。鉴于 Mac 操作系统本来是一个单用户系统，所以我们将关于对称多处理的更深入的讨论推迟到关于 Linux 的下一章，而该操作系统是从一开始就设计为支持多个用户并运行许多其他服务的。

另外，在第 8 版系统中还引入了个人网页共享，这一机制允许用户在自己的计算机上组织存储和发布网页。

5.10.1 多级文件系统升级版

伴随时间的行进步伐，硬盘驱动器的存储容量也变得越来越大。然而不幸的是，早先为了节省磁盘和随机存取存储器中的宝贵的空间，所以针对较小存储容量的驱动器而设计的文件系统所使用的是较小的指针。这些指针不能寻址较大存储容量的驱动器上的所有扇区，故而发明了各种机制来对早期的文件系统进行扩展以使其适应更大存储容量的驱动器。其中，第一种技术是按由多个扇区组成的盘块而不是单个扇区来进行分配。例如，在第 3 版系统中引入的多级文件系统在其数据结构中采用了 16 位指针，故而意味着只有 65 536 个扇区可以

① 在非对称多处理系统中，操作系统只会运行在某一个处理器上，而应用程序则可以运行在任何一个处理器上。尽管非对称多处理比对称多处理更为简单，但由于其限制了总的系统性能，所以现在这项技术已很少使用。

直接寻址处理。按照 512 字节的标准扇区大小,这便意味着相应文件系统不能支持存储容量大于 32MB 的驱动器。因此多级文件系统允许基于由多个扇区组成的盘块而不是单个扇区来进行分配。进一步说,如果分配是在两个扇区构成的盘块的基础上完成的,那么相同的 16 位指针就可以寻址 64MB 存储容量的驱动器。以此类推,这种方案可以按增加到 2 的任何整次幂数量的扇区来进行分配处理。就像苹果公司在 Mac 操作系统所引入的许多技术一样,这并不是一项新技术。它曾经用在了早期的 CP/M 系统中。需要指出的是,按较大盘块进行分配也存在一些问题。例如,在 1GB 的磁盘上,即使是一个 1 字节的文件也会占用 16KB 的磁盘空间(译者注:1GB/65 536=16KB,故需按 16KB 大小的盘块来进行分配)。如果使用了许多短小的文件,那么这种方式会变得非常低效,所以有必要设计一种新的文件系统来高效地寻址更大存储容量的驱动器。因此,8.1 版系统包含了一个多级文件系统的改进版,称之为多级文件系统升级版(Hierarchical File System Plus, HFS+)。该文件系统采用 32 位指针,所以能够直接寻址 4GB 个盘块,即存储容量为 2TB 的驱动器。若采用由 32 个扇区组成的盘块进行分配,那么这种文件系统可支持存储容量最高达 64TB 的驱动器。另外,多级文件系统升级版也支持长度为 255 字节的文件名。

5.10.2 其他的硬件变化

笼统地说是在计算机领域,确切地说是在 Mac 产品方面,相关硬件继续更新换代和不断发展。自从摩托罗拉把所有的开发工作都投入 PowerPC 线上之后,8.1 版系统成为支持 68K(即 68000)处理器的最后一版的 Mac 操作系统。另外,8.6 版系统增加了增强式电源管理,并改进了对诸如通用串行总线和火线(译者注:FireWire,一种高速输入/输出技术,用来连接外围设备与计算机,是高性能串行总线国际工业标准的苹果公司版本)等新的设备类型的支持。

为了允许单个应用程序可以使用多个处理器,8.6 版系统引入了允许应用程序将自身拆分为多个独立线程(在 Mac 操作系统中称为任务)的理念,在此基础上操作系统就可以在多个处理器上调度运行相关线程。我们将会在第 8 章深入地讨论这种技术。苹果公司对超微内核进行了修改以支持这种多线程。同时,有关操作系统还增加了对任务相关的优先级的支持,这便允许应用程序将一些任务指定为比其他任务更为重要。进一步说,如果某项任务正在等待某些事件且相应事件已完成,同时该任务的优先级要高于当前正在运行的任务,那么操作系统就可以通过停止正在运行任务并启动较高优先级的任务来实现处理器的抢占。我们在上一章的 Palm 操作系统中就曾见到了这项功能。然而,需要说明的是,在第 8 版系统中尚未发生进程抢占现象,换句话说,有关系统仍然在进程之间使用协作式多任务处理方式。

5.10.3 统一字符编码标准支持

在 8.5 版系统中,苹果公司开始支持一种新的机制,即利用一种所谓统一码(Unicode,或称为万国码、单一码)的全球化的字符编码标准来显示除英语之外的其他语言。与处理字符和字符串数据的旧的机制相比,统一码简化了使软件和其他语言协同工作的过程,即所谓本地化(localization)的过程。通过采用统一码来表示字符和字符串数据,对于每个可能的字符集而言,程序设计人员均可以仅仅使用单一的一套二进制文件,从而方便了数据的交换。统一码支持全世界各种语言所使用的众多文字,同时还涵盖许多技术符号和特殊字符。

统一码可以表示计算机使用中的绝大多数的字符。它提供了以下功能：

- 支持在一个文档中任意的组合源自于各种语言的字符组合。
- 对文字行为进行了规范和标准化。
- 为双向文本提供了标准化算法。
- 定义了到传统标准的映射。
- 定义了每个字符的语义。
- 定义了字符集的若干不同编码，包括 UTF-7、UTF-8、UTF-16 和 UTF-32。

统一码可以采用许多不同的方式加以使用，并且现在大多数的操作系统都在这个级别或者另一个级别上对统一码给予了支持。关于统一码的更全面的阐述可以在统一码联盟（Unicode Consortium，或称为统一码协会）的网站上找到：<http://www.unicode.org>。

5.11 第 9 版系统

到这个时候，Mac 操作系统的开发已经变得异常复杂。若干旨在构建新型操作系统的尝试得以启动，然后或者放弃或者出售给了曾经参与合作开发的公司。其中一个主要事件就是对 NeXT 计算机及其所附带的 NextStep 操作系统的收购。该操作系统将会最终演变成下一版本的 Mac 操作系统，即 X 版系统（System X，或称为第 10 版系统）。与此同时，Mac 操作系统的发布还得继续，故而在接下来的几年中，由一个被取消的操作系统项目所发明或改进的一些重要的功能陆续地被添加到了 Mac 操作系统中。这是对第 8 版 Mac 操作系统的持续稳步的推进，于是版本号从 8 向上递增而成为 9，从而为过渡到第 10 版系统铺平了道路。人们认为，数字之间的差距有可能会阻止某些用户从传统 Mac 操作系统迁移到 X 版操作系统（OS X，或称为第 10 版操作系统）上。第 9 版系统于 1999 年发布，且苹果公司称之为“有史以来最好的互联网操作系统”，当时，互联网的兴起已开始从多个方面对操作系统产生影响。

5.11.1 多用户

起初，个人计算机是被假定由一个人所使用的，而且 Mac 操作系统就体现了这种定位，例如一开始就没有像登录这样的机制。有关设计假定系统有唯一的一个用户，且如果安全存在问题，那么对相应机器的物理访问就限于那一个人。然而，许多因素纠缠到一起，逐渐削弱了这种假设。在工作场所，用户共享机器是很常见的事情，且有关用户往往只需要对机器进行短时间的访问。在家里，年轻的家庭成员总是想用计算机来玩游戏，但是现在他们开始重视访问互联网了，并需要利用软件来完成各种各样的作业，无论是写作、研究还是特殊应用程序的使用。他们还使用互联网来开展社交联系，从多人游戏、即时通信到聊天室。无论在家里还是在企业界，他们中的每一个人都在系统设置中拥有不同的偏好。相关设置包括图形化用户界面上的许多选项以及浏览器的主页等。他们还经常希望系统上有些文件——也许是个人日记——对其他人来说是无法访问的。因此，在第 9 版系统中添加了对多用户功能的支持。这要求每个用户在使用系统之前首先得登录。有关功能允许几个人共享一个 Mac 系统，同时保护他们的私人文件，并支持单独的系统和应用程序的首选项。Mac 系统通过多用户控制面板来进行设置和维护，允许一个用户为其他人创建账户，并授权相应用户可以正常访问或者受限访问应用程序、打印机或光盘驱动器。其间的多用户功能尚没有提供在更现代的操作系统或 X 版 Mac 操作系统（Mac OS X，也称为 X 版 Mac 操作系统，即 Mac OS X）

中可以找到的相同级别的安全性。进一步说,后来的那些操作系统具备文件的系统级别的安全性,而第9版系统并未提供此类安全性。例如,知识渊博的用户可以通过从另一不同的卷来引导从而实现受保护文件的访问。不过,这些多用户功能还是解决了 Mac 系统用户在共享机器时所面对的大量的长期存在的问题。

能够对某些用户的权限加以约束是一种安全稳妥的做法。但令人遗憾的是,许多用户对计算机不是很有经验,故而允许他们不受限制地访问却可能意味着他们很容易就会导致系统出现问题。对于最低限度而言,他们有可能会因为改变了什么地方,从而使他们不能正常地使用系统。在最坏的情况下,他们则可能让整个系统彻底崩溃,包括许多有价值的数据。好的做法认为,即使是知识渊博的用户通常也不应该以无限制的权限模式来运行系统。相反,他们应该在需要执行系统维护的时候使用特殊的管理登录方式。

密码是计算机系统中的—个长期的问题。鉴于不同的应用程序往往会拥有许多不同的密码和登录名,故而常常会导致用户采取一些不安全的做法,例如将有关密码和登录名写在便利贴上,然后将便利贴粘到显示器上。第9版系统则实现了一种被称为密码链访问的机制。这项功能对用户的多个标识符和密码进行管理,并给予安全存储。而且,一旦用户通过输入相应密码对密码链实现了解锁,那么密码链可感知的每个应用程序就能够从密码链数据库中获取特定应用程序的正确的用户名和密码,而无须再询问用户。

鉴于这时的文件保护并不十分安全,所以第9版系统还新增了文件加密功能。虽然加密方案非常健壮,但它是 Mac 操作系统专有的,因此以这种方式加密的文件只能由也运行第9版 Mac 操作系统的计算机来实施解密。如果 Windows 或 UNIX 计算机上的接收者需要解密相关文件,那么还需要一种跨平台的加密程序。但是,如果文件保护在特定的多用户情况下尚不够安全,那么加密机制无疑算是增加了一种安全措施。

5.11.2 网络

到20世纪90年代末,互联网取得极大成功,而 TCP/IP 已经成为所有个人计算机的一项要求。苹果公司是从第7版系统起为 TCP/IP 提供支持的,但仅限于某些功能,毕竟系统管理人员更喜欢和倾向于管理尽可能少量的不同协议。然而,由于 AppleTalk 并没有提供任何在 TCP/IP 中也未提供的主要功能,所以苹果公司承受了全面支持 TCP/IP 所有网络功能的巨大压力。因此,在第9版系统中,文件共享被修改为支持 TCP/IP。毕竟 AppleTalk 在互联网上没有得到支持,用户以前是无法通过互联网来远程访问在自己的 Mac 系统上正在处理的文件的,除非他们采取相关复杂、困难的技术。相应地,添加基于 TCP/IP 的文件共享支持则意味着 Mac 系统用户可以通过他们的标准互联网连接更加方便地在家工作。

108

此外,新的软件更新功能允许用户通过互联网来获取 Mac 操作系统的软件更新,并在有关更新可用的时候通知用户。这大大简化了系统管理员的工作。

5.11.3 应用程序接口

当第9版系统还在开发的时候,X版操作系统就已经起步了。正如我们很快将会看到的那样,X版操作系统本质上是一种不同的操作系统。然而,苹果公司并不希望它被这样认为。因此,让许多旧的应用程序仍然可以在新的操作系统上执行是至关重要的。我们已经讨论过从68000处理器架构过渡到 PowerPC 处理器架构期间所需完成的仿真。类似地,在新的操作系统下执行大多数旧版的应用程序接口也是可行的,但是,如果是通过修改旧的应

用程序来支持 X 版操作系统中所提供的应用程序接口，则并不可取。为此，苹果公司为第 9 版系统创建了一套新的应用程序接口，使其向前兼容 X 版操作系统，同时包含了对大多数旧式的应用程序接口函数的支持。这一新的应用程序接口被称为碳式应用程序接口（Carbon API），它包括对大约 70% 的传统 Mac 操作系统应用程序接口的支持。

5.11.4 视频

计算机游戏是为个人计算机开发先进强大的视频功能背后的驱动力之一。虽然像桌面排版等其他应用程序也可以受益于这些功能，但是有更多的人是在玩游戏，而不是使用系统来完成桌面排版。自然而然地，硬件厂商希望为更大的市场开发产品。苹果计算机公司也不例外，故而 Mac 系统提供了许多游戏。在第 9 版系统中新增加的一项功能是对内置在硬件里用来支持三维对象加速渲染的视频卡以及对诸如 OpenGL（译者注：Open Graphics Library，一套三维图形处理库，也是该领域的工业标准）等用来改进视频和游戏体验的相关技术的软件应用程序接口的支持。

5.12 X 版 Mac 操作系统

在操作系统发展史上，X 版操作系统可能是最具创新性的变化之一，而这不仅仅是因为苹果公司将版本命名从 System 10（第 10 版系统）改成了 OS X（X 版操作系统，即第 10 版操作系统）。更值得一提的是，在 X 版操作系统中，苹果公司完全抛弃了第 9 版系统内核，并将其替换为另一种内核。与此同时，微软的 Windows 3.x 自 1990 年发布以来取得了巨大成功，随后，微软在 1993 年还发布了另一个成功的操作系统，即 Windows NT。而 NT 是为高端应用程序设计的高级操作系统，包括诸如抢先式多任务、支持运行多种传统操作系统应用程序的能力、多处理器支持以及一种新型文件系统等各项功能。苹果公司需要一种能与这些微软产品相匹敌的操作系统。如前所述，苹果公司曾经与不同公司在多个操作系统项目中有过合作，但没有一个项目成功提供他们所需要的操作系统。他们还曾考虑过在 Solaris 操作系统（开发厂商为太阳微系统公司，即 Sun Microsystems）、BeOS 操作系统（开发厂商为 Be 公司），据传言甚至还有 NT 操作系统（开发厂商为微软）的内核之上来构建新的操作系统。不过，他们最终选定了建立在 Mach 内核和 UNIX 衍生版 FreeBSD 基础之上的微内核（译者注：Mach 是由卡内基梅隆大学开发的用于支持操作系统研究的操作系统微内核，而 FreeBSD 则是由加州大学伯克利分校开发的 UNIX 版衍生形成的免费开源的操作系统），同时这二者也是由 NeXT 计算机公司所开发的面向对象的 NextStep 操作系统的基础。出于性能原因的考量，一些 FreeBSD 代码与 Mach 内核进行了整合，故而其结果并不是真正的微内核。X 版操作系统的确切演变过程追溯起来比较困难，并且与本教材不是非常关联，所以对于那些对各种不同意见感兴趣的人，建议可以到万维网上去查找更多大量的相关信息。

在第 9 版操作系统软件中所做的更改，允许在 X 版操作系统中引导进入传统环境（classic environment）。因此，传统环境实际就是 X 版操作系统的一个应用程序，其提供了一个可以运行第 9 版操作系统的兼容层级，允许传统应用程序无须移植到新的应用程序接口上就可以运行在 X 版操作系统上。这种设计相当完美，不过传统的应用程序依然保持着它们原先的第 8 版或第 9 版操作系统所支持的外观和式样，看起来并不像 X 版操作系统的应用程序。

5.12.1 新功能

为此，X 版操作系统实际上是一种不同的操作系统，尽管其支持以往在传统 Mac 操作系统版本中所使用的应用程序接口。进一步说，X 版操作系统的许多功能源自于 UNIX 实用程序软件包。在下一章中，我们将深入探究另一种 UNIX 衍生版操作系统。在此，我们只简单罗列一下 X 版操作系统带给 Mac 世界的一些特性：

- 新的内存管理系统，允许更多的程序同时运行，并支持完全内存保护，可防止程序彼此间相互影响和搞瘫对方。
- 命令行（UNIX 终端仿真的组成部分）。
- 在进程之间而不是仅限于线程之间的抢占式多任务。
- 支持 UNIX 文件系统格式。
- 阿帕奇网络服务器（Apache Web server）。
- 全面支持对称多处理。

5.12.2 又一款新处理器

鉴于 X 版操作系统的更强大的功能对系统资源提出了更高的要求，所以 PowerPC G3 处理器是该版操作系统的最低要求。

2005 年 6 月，苹果计算机公司宣布他们将把 Mac 产品线从 PowerPC 处理器转移到英特尔产品上。2006 年 1 月，苹果计算机发布了第一款采用英特尔处理器的 Mac 计算机。传统（仿真）环境在 x86 平台的 X 版操作系统中无法运行。另外需要说明的是，大多数编写得很好的“传统”应用程序在该环境下均可正常运行，然而兼容性只有在有关软件没有直接与硬件进行交互且仅仅与操作系统应用程序接口打交道的前提条件下才有保证。

110

5.13 小结

在本章，我们讨论了一种更复杂的现代操作系统——由苹果计算机公司所开发的 Mac 操作系统——的相关功能特征和概念。该操作系统的开发是为了向市场推出一种带有图形化用户界面的廉价的个人计算机。其通常情况下仅支持单个用户，但后来的版本允许多个进程同时运行，并支持用户应用程序启动多个线程。我们从第 5.1 节对 Mac 操作系统的概述出发，开始了本章的讨论。我们在这一章中采用了一种不同的叙述方式，即沿着 Mac 操作系统的各个版本，依次描述了每个版本中主要的新功能。这是因为 Mac 操作系统最初是一个非常简单的系统，除了图形化用户界面之外，没有比 CP/M 提供任何更多的功能，跟我们现在的想法相比，甚至可以说是非常原始。

然而，Mac 操作系统却最终演变发展成为一个现代的、功能齐全的、能够支持多个用户和多个进程的操作系统。我们以 X 版 Mac 操作系统的简要说明结束了有关传奇历程。下一章我们将阐述另一种多用户系统，即 Linux 操作系统。

参考文献

Apple Computer, *Inside Macintosh series*. Pearson Professional Education, 1992.
Danuloff, C., *The System 7 Book: Getting the Most from Your New Macintosh Operating System*. Chapel Hill,

NC: Ventana Press, 1992.
Lewis, R., and B. Fishman, *Mac OS in a Nutshell*, 1st ed. Sebastopol, CA: O'Reilly Media, 2000.

网上资源

<http://applemuseum.bott.org> (局外人眼里的Mac操作系统的发展史)
<http://developer.apple.com/documentation/> (包含各种链接指向Mac内部资料系列文档, 可按PDF格式下载)
<http://developer.apple.com/technotes/>
<http://www.apple-history.com> (局外人眼里的Mac操作系统的发展史)
http://www.macos.utah.edu/documentation/operating_systems/mac_os_x.html
<http://www.online-literature.com/orwell/1984/> (“1984”

电视广告背后的著作)
<http://rolli.ch/MacPlus> (指向vMac的链接)
http://en.wikipedia.org/wiki/Mac_OS_history
 (局外人眼里的Mac操作系统的发展史)
<http://en.wikipedia.org/wiki/NuBus> (最初的Mac总线)
<http://en.wikipedia.org/wiki/Y2k> (关于“千禧虫”的解释)
<http://www.parc.xerox.com/about/history/default.html>
http://en.wikipedia.org/wiki/Mach_kernel (X版Mac操作系统的内核)

习题

- 5.1 下面哪种操作系统是第一个使用图形化用户界面的系统？
 - a. Xerox Star
 - b. UNIX X Windows
 - c. Xerox Alto
 - d. Apple Lisa
 - e. 以上选项均非第一个使用图形化用户界面的系统
- 5.2 苹果公司的 Mac 计算机是在 IBM PC 之后开始创建和引入的，而且要比后者稍微便宜一些。这是否正确？
- 5.3 Mac 系统使用哪种处理器呢？
 - a. 摩托罗拉 68000 系列
 - b. 摩托罗拉 PowerPC 系列
 - c. 英特尔 80x86 系列
 - d. 以上均不对
 - e. 以上都是
- 5.4 与大多数其他个人计算机操作系统相比，Mac 系统拥有的巨大优势是什么？
- 5.5 苹果公司的 Lisa 系统是 Mac 系统的前身，并且可以同时运行多道应用程序。最初的 Mac 系统一次运行多少道应用程序？为什么？
- 5.6 最初的 Mac 系统不支持可以防止应用程序破坏操作系统或其数据的内存保护。这是否正确？
- 5.7 最初的 Mac 操作系统支持的文件夹（目录）层级是多少？
- 5.8 68000 处理器一次能够寻址多大的内存空间？
 - a. 16KB
 - b. 64KB
 - c. 128KB
 - d. 1MB
 - e. 68000 处理器在任何时候都能够访问所有的内存
- 5.9 在 Mac 操作系统中，内核以管理程序模式运行，而应用程序则在用户模式下运行。这是否正确？
- 5.10 在 Mac 操作系统中实现应用程序栈和堆的方式存在什么困难？
- 5.11 Mac 操作系统是如何避免上一道习题中所涉及的问题的？
- 5.12 堆内存管理的方式所引发的问题是什么？Mac 操作系统是如何解决的？
- 5.13 关于堆管理问题，Mac 操作系统的解决方案与 Palm 操作系统有何不同？
- 5.14 与大多数其他个人计算机操作系统不同，Mac 操作系统将大部分操作系统本身都存放在只读存储器中。这是为什么呢？

- 5.15 对于早期版本的 Mac 操作系统来说, 剪切和粘贴操作通常需要几分钟而不是几秒钟。有关操作系统的哪项新功能改变了这一点呢?
- 5.16 在习题 5.15 中所提到的变化是否使 Mac 操作系统成为一种多任务操作系统呢?
- 5.17 关于多级文件系统, 引入了哪些主要变化呢?
- 5.18 多重查找器 (MultiFinder) 是做什么的?
- a. 它允许用户在文件中搜索多个字符串
 - b. 它允许多个用户登录到系统
 - c. 它允许用户在网络中搜索其他用户
 - d. 它像今天的谷歌一样搜索互联网
 - e. 以上均不对
- 5.19 第 5 版系统提供了哪些值得关注的新功能?
- 5.20 第 6 版系统支持采用 32 位寻址模式的 Mac 新机型, 由此引发了什么问题呢?
- 5.21 “胖二进制代码”有什么用?
- 5.22 虚拟内存使用软件来模拟所缺失的内存块。这是否正确?
- 5.23 多线程的主要用途是什么?
- 5.24 在第 9 版系统的各种版本中添加了很多的增强功能, 请列举出其中的三项。
- 5.25 我们为什么没有过多地阐述 X 版 Mac 操作系统?

多用户操作系统

在这一章，我们将讨论一种比上一章所论述的 Mac 操作系统——至少比 X 版操作系统之前版本的 Mac 操作系统——功能更为强大的操作系统。这就是 Linux 操作系统。本章专注的要点是那些根据操作系统的多用户需求而应当包含的相关附加功能，而不是去讨论 Linux 操作系统的^[113]所有方面。我们在第 19 章将再次就 Linux 操作系统展开进一步的讨论，着眼于从一个更为完整的实例研究的角度来审视其用来支持主要系统模块的各种机制所做的决策。

在本章第 6.1 节，我们将首先概要介绍 Linux 操作系统及其历史背景。在第 6.2 节中，我们将讨论多用户操作系统的本质以及这一设计决策是如何影响到操作系统的相关功能的。接下来在第 6.3 节，我们将阐明 Linux 操作系统中的进程和任务的调度。我们已经在其他操作系统那里见识到一些这方面的功能，但是 Linux 操作系统是我们研究的第一个完整地实现了进程和线程的所有概念的操作系统。最后，我们将在第 6.4 节对全章内容进行归纳总结。

6.1 引言

Linux 操作系统的设计是以 UNIX 操作系统为基础的。UNIX 操作系统是一种早期的操作系统，最初开发主要用于支持远程终端（常常是带有串行数据电缆的显示屏和键盘）上的多个用户。这些终端与中央计算机系统相连接，甚至可能通过了调制解调器和电话线。换句话说，UNIX 操作系统最初创建是为了给大型计算机开发环境虚拟一个不那么昂贵的小型计算机。（相关开发被其两位首创人员作为某种业余爱好，而他们二人则因为 UNIX 的概念和设计机理而赢得了非常著名的计算机领域大奖，即图灵奖。）还有一些版本的 Linux 操作系统，是着眼于许多其他的应用场景。归纳而言，有关系统的设计目标包括：

- 支持在个人计算机控制台上的单个用户。
- 担当用于各种远程访问功能（例如文件、打印和目录服务等）的服务器。
- 用作诸如数据库管理系统、超文本传输协议（Hypertext Transport Protocol, HTTP）或网站（Web）服务器以及文件传输协议（File Transfer Protocol, ftp 或 FTP）服务器等其他高级服务的平台。
- 充当网络中的路由器。
- 控制实时系统。
- 嵌入在没有直接的人类用户的设备中。

6.1.1 多用户操作系统的历史

Linux 是在 UNIX 的启发下诞生和发展起来的，所以在介绍 Linux 操作系统之前，先简要讨论一下 UNIX 操作系统的起源是很有意义的。1969 年，贝尔实验室（Bell Laboratories）的肯·汤普逊（Ken Thompson）开始尝试利用一台被废弃的 PDP-7 小型计算机来创建一个

多用户、多任务的操作系统。他与丹尼斯·里奇 (Dennis Ritchie) 合作, 然后他俩和研究小组的其他成员一起创作了 UNIX 操作系统的第一个版本, 当时称为 Unics, 寓意为对他俩都曾参与的 Multics 项目的一次挖掘。(Multics 是一个庞大的项目, 有一百多人参加相关研发工作, 但只有少数几个程序设计人员参与创建了 UNIX。)早期版本的 UNIX 是用汇编语言编写的, 但第三个版本则是用 C 编程语言编写的, 且 C 语言是由里奇特意作为编写操作系统的程序设计语言而精巧设计的。进一步说, C 被设计为一种非常低级且简单的语言, 允许程序设计人员在大多数情况下忽略许多硬件细节, 但是仍然以编译器可以利用特殊硬件特性的方式来编写程序。UNIX 是开发它的贝尔实验室的母公司, 即美国电话电报公司 (AT&T) 的专利产品。而美国电话电报公司对特许使用 UNIX 开展学术研究仅收取适度的费用。UNIX 第 6 版 (大约在 1976 年) 对大学是完全免费的, 而到了第 7 版也仅仅收费 100 美元。有关许可费用包括所有的源代码, 并可随意修改。不过, 对于政府实验室和商业实体来说, 则必须支付 21 000 美元。这对于当时一台机器动辄数十万美元甚至几百万美元的费用来说, 对操作系统而言也算不上什么不合理的价格。然而对于大学而言, 学术许可证是一笔不可抗拒和非常诱人的交易, 因为那里有对知识和技能充满热切渴望的学生, 而且那些学生可以把 UNIX 移植到其他机器上或者按照自己的合理想法对其进行改进和提高。对于那些通常与操作系统一起分销的实用程序如文本编辑器之类的东西来说, 尤其如此。

作为一种简单、一致、短小 (运行在几千字节的内存空间, 源代码只有几千行且大部分是 C 代码) 却非常灵活的操作系统, UNIX 的魅力是无法抗拒的。不少公司和研究团队纷纷仿照 UNIX 编写类似系统, 采用与 UNIX 相似的工作机理和运行机制、具有相同的操作系统系统调用和操作系统实用程序, 但源代码却完全重写 (以规避美国电话电报公司的所有权以及排除获得美国电话电报公司批准的需要)。

114

1991 年, 芬兰赫尔辛基大学 (University of Helsinki) 计算机专业学生李纳斯·托瓦兹 (Linus Torvalds) 通过课堂作业熟悉了 UNIX, 并因而想找一种类 UNIX 操作系统在家里使用。MINIX 是为数不多的免费选项之一 (一本教科书附带), 由安德鲁·塔嫩鲍姆 (Andrew Tanenbaum) 出于教学目的而编写, 类似于 UNIX 但功能极其有限。当然, 也有其他的一些类似 UNIX 的免费操作系统, 但大多数尚不成熟或不太稳定, 或者要求比大多数用户在家里所用机器更高端的硬件条件。当托瓦兹使用 MINIX 的时候, 发觉该系统少了许多功能, 所以他决定重写 MINIX。一开始, 他保留了原先的文件系统设计方案, 但后来替换成了自己的方案。MINIX 运行在一个非常基本的 8088 处理器和软盘上, 从而使其可以运行在非常廉价的硬件系统上。不过, 它却没有利用新型处理器和硬盘的功能。托瓦兹使用了一台基于英特尔 386 处理器 (Intel 386) 的个人计算机, 刚开始时是添加功能, 但最终却写成了一种新的操作系统, 且相关开发最初是使用 MINIX 系统上的 C 编译器来完成的。不久之后, Linux 已经成为一个“真正的”操作系统。特别地, 所生成的 Linux 内核不再包含任何 UNIX 或 MINIX 的代码。准确地说, Linux 是在 UNIX 接口及其实用程序基础上的完全重写。实际上, Linux 只是一个操作系统的内核。它的构建利用了位于马萨诸塞州剑桥的自由软件基金会 (Free Software Foundation) 成员所开发的 GNU 软件 (译者注: GNU 的全称为 GNU's Not UNIX, 即 GNU 并非 UNIX, 其目标是创建一套完全免费的操作系统, 并推行软件的自由使用、复制、修改和发布), 同时还把大量的 GNU 软件作为相应的实用程序和应用程序, 从而确保构成一套完整的操作系统。事实上, 操作系统的内核以外的主要部分也是 GNU 计划的组成部分。因此, Linux 系统更有意思、更为重要的特性之一是其并不专属

于一家公司。而在此之前我们所讨论的所有的操作系统都是由特定的一家公司所拥有的，所以他们会认为源代码属于商业秘密，且在一般情况下并不向公众发布。但是，Linux 和 GNU 软件都是“开源”项目^①。相关源代码是免费提供的，而且鼓励用户去纠正错误和增强代码。另外，是专利性质的过程比开源模式的过程开发的操作系统更好更健壮，还是刚好相反，关于这点目前存在着广泛的争论。

尽管根据有关准确说明，Linux 提供了一套免费的支持 UNIX 操作的操作系统版本，然而这并不是一条像其表面看起来那样清晰或令人满意的声明。一方面（一定程度上是因为 UNIX 的源代码针对大学几乎是免费的），在 UNIX 开发的历程中，相关衍生版本一直比比皆是。许多着手将其移植到另一种环境的程序设计人员往往无法抵制去“改进”某样东西或者添加某项最爱功能的诱惑。直到 20 世纪 80 年代末期，才由独立的电气电子工程师协会（Institute of Electrical and Electronic Engineers, IEEE）创立了一套相当标准的 UNIX 应用程序接口，并称之为 POSIX（Portable Operating System Interface of UNIX，UNIX 可移植操作系统接口）标准。令人遗憾的是，电气电子工程师协会针对该标准的使用收取大笔的费用，从而导致类 UNIX 操作系统的免费版本的开发人员通常无法付得起费用和让他们的产品被电气电子工程师协会认证为是符合 POSIX 标准的。为此，后来又产生了另外一套规范，即对于小公司或未付费开发人员更容易使用的单一 UNIX 规范（Single UNIX Specification, SUS）。

115

在 Linux 操作系统最初可供使用的时候，一个想要成为 Linux 用户的人需要是某种意义上的 UNIX 专家，其知道需要什么库和可执行文件能够成功地让 Linux 引导和运行起来，同时还要了解配置相关的细节以及系统中某些文件的位置。不过，许多潜在的用户只是希望该系统可用于他们的工作、爱好或研究，对钻研内核并不感兴趣，也无意成为从头开始构建系统的专家。与此同时，Linux 源代码是免费的，故而任何人都可以制作系统的副本并销售相应副本。因此，个人、大学和公司纷纷开始创建 Linux 操作系统的发行版本（distribution）。Linux 发行版通常包括经过编译处理的 Linux 内核、GNU 系统库和实用程序的可执行代码版本。许多发行版还提供了一套安装程序，这与其他操作系统为给定机器定制相应操作系统所提供的安装程序相类似。这种发行版最初只是为了方便，但是因为节省时间，同时也可降低忽略（从源码出发构建系统过程中的）一些小而重要的细节的概率，所以，今天它们已经成为通常的安装方法，即便是对于 UNIX 或 Linux 大师也是如此。现在，大多数 Linux 发行版都被认证为是符合单一 UNIX 规范的。同时，还有许多不同的 Linux 发行版是专为特殊目的而设计的，例如从硬盘驱动器以外的设备来启动、使用 Linux 作为服务器或者支持不同的语言作为默认设置等。

Linux 系统的管理本身也是一个很有意思的话题，而各种版本的编号则是 Linux 的关键特性之一。具体而言，主版本号（major release number）是其第一个整数。托瓦兹首次发布的初始版本是版本 0，而当前版本是 2（译者注：在本翻译教材出版的时候，Linux 主版本号已上升为 4）。对于版本编号的第二部分而言，如果相应整数是奇数，就表示是开发版本（有时称为“黑客”版本），而如果是偶数，则表示是产品版本（有时称为“用户”版本）。此外，还添加有另外一个整数用来区分各种各样的补丁级别。

Linux 操作系统确实远远超越了它非常简陋的开端。刚开始作为一个操作系统内核时，

① “开源”许可证的概念有许多变种或版本，各种版本的拥护者通常对相应变种抱持坚定不移的态度。我们在这个术语则按照一种宽泛、一般的含义来加以使用。

它只能够用在单处理器架构下的英特尔 386 处理器（或更好的）系统上。然而现在，它可用在几乎所有可用的硬件平台上，包括在许多情况下硬件供应商提供的专有操作系统有时甚至是某版 UNIX 操作系统的平台上。（当然，一些实现方案会比其他的实现方案更好。）例如，IBM 公司已经非常热情地接受和采用了 Linux 操作系统，并将 Linux 移植到其所有 4 个 E 系列的系统线上。这一策略利用了 Linux 应用程序的可移植性。目前，IBM 公司的收入的很大一部分来自于应用程序的编写、安装和支持，而不是销售硬件或操作系统。他们很可能经常发现自己处于应用程序的创建与移植这一循环状态中——在他们的 4 个硬件产品线中的某个硬件产品线上创建一个应用程序，然后便不得不把该应用程序移植到其他的平台上供其他客户使用。通过 Linux 和 Java，他们就可以一次性创建应用程序，并轻松地将它们（包括利用 Linux 软件包、脚本等在内的所有安装和支持程序）迁移到其他平台。

6.1.2 Linux 操作系统的基本结构

Linux 操作系统使用的是整体式内核（monolithic kernel，或称为大内核）。这意味着整个内核被加载到一个包含操作系统所有模块的单个程序中。每个模块都可以直接访问内核中的任何函数、对象或数据结构。但这也意味着整体式操作系统的运行效率常常比微内核操作系统要更为高效。当然，这种方法也存在若干隐患和风险。首先，由于所有的操作系统代码都运行在管理程序模式，所以任何错误在理论上都可能会导致较为严重的问题。其次，因为与特定机器相关的部分不一定被很好地进行了隔离，所以要移植到新的体系结构上通常比较困难。再次，如果有关设计人员不太注意，源代码可能很快就会变得异常复杂，因为这种内核与微内核相比，在各模块之间并不一定会拥有清晰的、明确定义的接口。另外，对于整体式内核来说，添加对新设备的支持往往也比较困难。通常情况下，需要编译新的驱动程序，并重新链接和重新加载内核。显然，这意味着操作系统必须得停下来和重新启动——在多用户系统或者提供许多网络服务、服务于许多用户或兼有这两种情况的服务器中都是不乐意见到的事情。不过，正如我们很快将会看到的那样，现代 Linux 操作系统版本已经克服了许多这样的问题。Linux 内核的结构如图 6-1 所示。

[116]

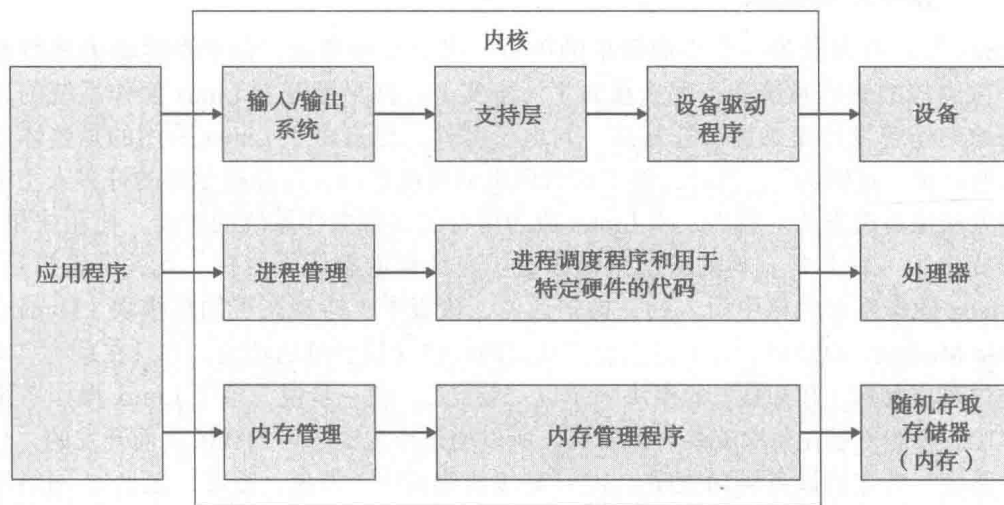


图 6-1 Linux 系统架构

如前所述，操作系统的另一种类型的组织是建立在微内核（microkernel）上，对应结构

如图 6-2 所示。同样，这意味着内核相关代码已经被进行了最小化处理，即仅仅包含那些绝对必须在内核以便于执行特权指令的代码。一般情况下，有关代码包括进程管理、基本的内存管理以及进程间通信。而那些剩余的我们通常认为也应当作为常驻内存操作系统组成部分的功能则可能在用户模式下运行。这种组织方式有一些好的地方，但也会付出一定的代价。进一步说，这种结构更易于构建一个健壮性的内核，并且更容易将其移植到一个新的平台上。同时，这种结构的主要代价是它经常会引进更多的开销——中断处理和上下文切换常常会使操作系统运行速度比起整体式内核要慢些。MINIX 就是按照微内核系统结构进行设计和构建的。

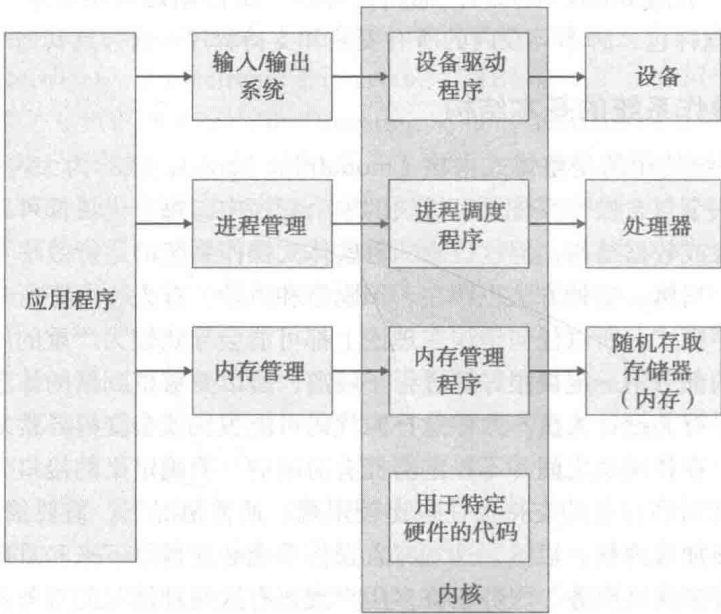


图 6-2 一个微内核系统架构

6.1.3 动态可加载模块

Linux 最初被设想为一个小而简单的项目。出于这种考量，似乎费尽心力地特意去创建一个微内核的操作系统就不那么重要了。事实上，在早期开发 Linux 操作系统的某个时间，塔嫩鲍姆曾发给李纳斯·托瓦兹一封电子邮件，坦言由于 Linux 采用的是整体式内核方法，所以是“过时的”。当时，鉴于前面列出的那些理由，计算科学界的许多人士都认为微内核方法是首选方法。然而，当 Linux 成为可行的备选操作系统的时候，托瓦兹和 Linux 社区却提出了一种非常有趣的方法来修改或扩充纯粹的整体式内核。其关键思想是在 2.0 版的 Linux 操作系统内核中引入的，换句话说，该版本支持**动态可加载模块**（Dynamically Loadable Module, DLM）。有关设想允许基本内核包含最少量的功能，并且在系统启动运行之后通过可以加载（和卸载）的模块来予以“装点”。进一步说，鉴于 Linux 操作系统的许多基本功能可能不是在每次安装时都需要，所以均是作为动态可加载模块而开发的，具体包括文件系统、特定的设备驱动程序、SCSI 高级驱动程序（磁盘、磁带、光盘）、网络驱动程序、行式打印机驱动程序以及串行（电传打字机终端）驱动程序等功能。

为了支持动态可加载模块，核心内核必须设立明确定义的接口。这样做便消除了对整体式方法存有异议的一项重要理由。当一个模块被加载时，它将调用一个操作系统函数把自己

[117]

“注册”到内核，而要调用的确切函数则取决于要加载的模块类型，具体可参见表 6-1 关于此类调用所给出的一组解释说明。

118

表 6-1 动态模块注册函数

目 的	动态注册函数	目 的	动态注册函数
模块	init-module	串行接口	register_serial
符号表	register_symtab	文件系统	register_filesystem
控制台驱动程序	tty_register_driver	二进制格式	register_binfmt
传输协议	inet_add_protocol	块设备	register_blkdev
网络协议	dev_add_pack	字符设备	register_chrdev
链接协议	register_netdev		

关于动态可加载模块接口的一个令人关注的作用是，它允许软件开发人员可以在不提供源代码（根据各种开源许可证，提供源码则是必要的）的情况下为 Linux 系统创建增强功能。这样就能允许 Linux 依然是开源项目，但同时也可包含由开发人员保留为专有的功能函数。

关于动态可加载模块的另一要点是，它们需要与核心内核函数及数据结构进行链接（也就是说，它们需要能被内核找到，并且反过来它们也需要能够访问内核的相关部分）。这是通过将符号表（symbol table）作为内核的一部分进行加载来实现的，该表称为 ksym。任何将会在内核中公开的函数或数据结构均需在此符号表中拥有一项定义。而模块在加载时将会调用一个函数来搜索符号表并解析正在加载的模块中的所有引用。这听起来好像会降低系统的运行速度，但是模块通常只会加载一次，然后就保持成为系统的一部分。即使它们被重复地添加和移除（例如对于可移动通用串行总线设备来说），但与处理器速度相比，一般情况下相应的间隔时间或许还是比较长的。

也有可能是欲被内核加载的模块想要公开其自身的函数和数据结构。在这种情况下，则可以利用一个简单的功能函数 EXPORT_SYMBOL 把相关表项添加到符号表中就可实现相关目的。

6.1.4 中断处理

如前所述，Linux 操作系统中的设备管理是中断驱动的。硬件中断是指相关硬件可以把异步事件（例子如一个数据包到达网络适配器的事件）通知操作系统的一种机制。具体而言，当有关适配器接收到数据包时，将会产生中断，从而使操作系统可以停止其正在完成的事情并转去处理刚刚到达的数据包。有些时候，处理有关数据包所需的处理时间可能相当冗长。或者，比起系统当时正在进行的操作，数据包的完整处理可能并不那么重要。尽管如此，有一些最起码的工作确实需要由内核立即完成。换句话说，操作系统可能至少需要为可能到达的任何额外的数据包分配新的缓冲区。当这项工作正在完成的时候，通常情况下，要么所有级别的中断均已禁用，要么当前中断级别以及任何较低优先级的中断已被禁用。当然，让中断禁用很长时间或者让一些外部事件错失并不是一个好主意。因此，Linux 操作系统的中断处理程序采用了一种众所周知的、流行的上下半部有机结合的组织结构。上半部分（top-half）由那些需要立即发生的事情组成，而下半部分（bottom-half）则是那些能够以更宽松的时限来完成的事情。同时，上半部分将会记录足够的信息，以方便下半部分可以稍后完成相

119

关工作。在 Linux 的后续版本中，下半部分的结构被重新设计并被赋予了一个新的名称——小任务 (tasklet)。之所以重新设计，其主要原因在于，对于多处理器环境而言，小任务可以运行在多个处理器上，而原先的下半部分结构每次只能由一个处理器加以执行。简而言之，已有的下半部分组织结构大部分都顺应这种变化而进行了重新设计。

6.1.5 文件系统目录树

与 UNIX 一样，Linux 操作系统具有很强的围绕文件系统的导向。许多根本就不是文件的东西也会出现在文件系统树中，如图 6-3 所示。其中，目录树的根部显示在最顶层。注意，无论所谓的 `proc` 目录还是所谓的 `dev` 目录实际上都不是目录。进一步说，它们分别表示正在运行的进程以及系统上的硬件（或虚拟）设备。对这些名称的引用将会引发 Linux 操作系统去调用其他的函数，而后者将会返回关于这些元素在被访问时的适当的信息。关于这些内容具体将在第 19 章展开进一步的讨论。另外，在图 6-3 中还可以看到的其他令人关注的目录是 `/home` 目录下的子目录，它们是对应于各个用户的目录。当一个用户登录进入 Linux 系统的时候，操作系统将会把该用户的主目录设置为当前工作目录。

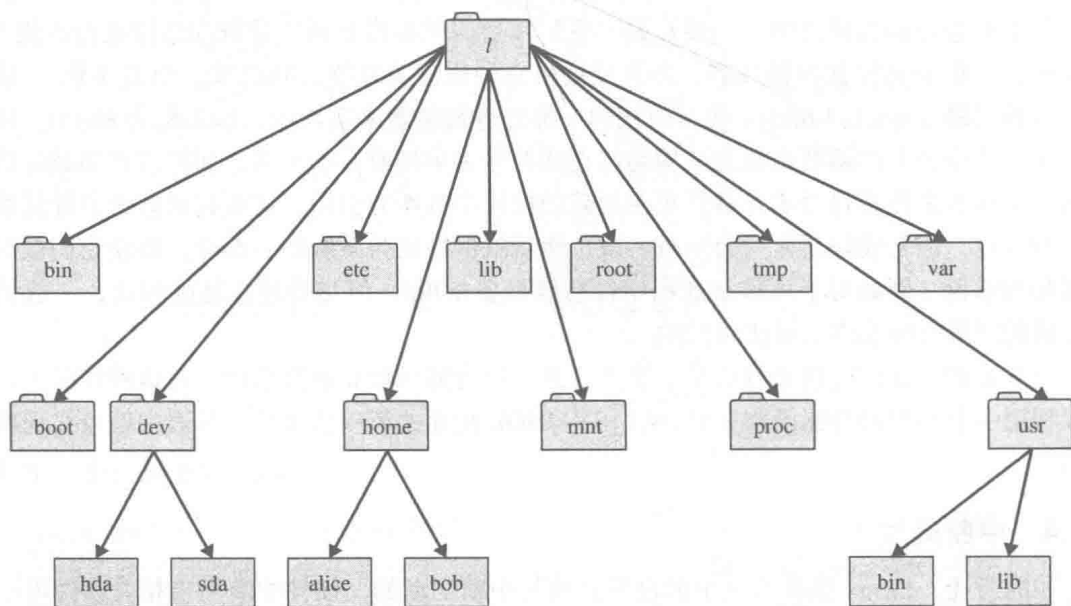


图 6-3 Linux 操作系统目录树的一部分

6.2 多用户操作系统环境

由于 Linux 是仿照 UNIX 而设计和开发的，且 UNIX 操作系统是多用户系统，所以 Linux 操作系统也是一个多用户系统。关于系统存在多个用户的假设，从一开始就引进了一个直到现在我们都不必太过担心的问题——信息安全。当只有一个用户可以使用一台计算机的时候，有关操作系统通常不需要关心对应用户访问相应计算机上的任何文件的权限。其一般假定该计算机的任何用户可以访问任何文件，并且通过限制对相应机器的访问或者通过利用有关操作系统外部的实用程序对文件进行加密和实施保护来提供文件的安全性。但是，系统上的多个用户往往会同时要求操作系统提供一种机制来保护每个用户的文件免受其他用户的影响。这意味着有关操作系统应当需要知道当前用户是谁。自然地，这也意味着用户需要

使用一个用户标识符 (identifier, ID) 和一个密码登录到计算机上。当然, 有时相关用户会希望共享文件, 故而有关操作系统需要提供允许一些文件共享的相关机制。所有的多用户系统常常也会用作服务器, 并且可能有多个用户通过远程来进行登录。因此, 这些操作系统也拥有安全特性, 具体将会在后面的章节展开讨论。还有, 就像我们在 Mac 操作系统中所看到的那样, 当计算机添加到网络中的时候, 即使是单用户系统也需要提供保护各种资源的相关机制, 所以用户登录等已经成为当前大多数操作系统的常见功能, 即便仅仅是为了网络访问。此外, Linux 操作系统的服务器版本允许多个用户远程访问系统上的文件和其他资源。尽管这不是 Linux 操作系统的主要方面, 但是同时运行许多服务和许多用户应用程序的能力意味着其还必须为诸如多道程序和多线程等高级特性提供相应的支持。支持多个用户在这一领域并没有引入任何新的需求, 但是 Linux 操作系统确实针对这一问题采取了一种不同的方法, 尤其是从它的 UNIX 渊源的角度来考虑的话。

6.2.1 文件访问权限

Linux 操作系统与其他类 UNIX 系统支持相同的文件保护和共享模型。对于任何特定的文件, Linux 操作系统都将把所有用户视为三个集合中的某一个集合的成员。其中, 第一个集合只有一个成员, 即文件的所有者。最开始在创建文件的时候, 文件的所有者就是创建该文件的那个人。第二个集合是由系统管理员 (system administrator 或 sysadmin) ——对应人员一般就是被这样称呼的——预先定义的一个集合。该集合通常是对一组文件有着共同兴趣的一群用户。或许就是一个项目团队, 他们正在努力开发一个新产品的相关文档或者正在使用相同的源代码, 并希望在团队成员之间进行共享。于是系统管理员就可以按名称标定一个新的组别, 并把对应用户指定为该组的成员。第三个集合是“每个人”, 但实际上指的是每个均非其他两个集合成员的用户, 即“其他人”。对于每个集合的成员而言, 针对一个特定的文件, 可以允许三种类型的访问方式: 读、写和执行。

文件的所有者可以利用一个名为 `chmod` 的实用程序来设置文件的访问权限。这条典型的令人费解的 Linux 命令代表的是“更改文件模式”。该实用程序接收两个参数, 即一个文件名和一个“模式”, 后者用于指定对文件模式 (file mode) 所做的更改。传统上, 有关模式是一个三位数, 且该数的每个数位均限制为一个八进制数字, 也就是说, 数位的取值范围为 0~7。进一步说, 每个八进制数字又可以被认为是由三个二进制位组成的, 而这三个二进制位分别用于准许各种操作——读、写和执行。同时, 模式的三位数分别给出三个集合即所有者、组别和其他人所拥有的访问权限。用来罗列目录内容的 `ls` 命令, 可以列出一个文件或目录的相应模式设置。为此, 请考虑如下的由 `ls` 命令显示的结果条目:

```
-rwxr-x--x gil develop spellcheck
```

这一条目描述的是名为“spellcheck”的可执行文件。该行的第一部分是访问权限的设置。最前面的“-”具有其他用途。最开始的模式“rwx”适用于文件的所有者, 在本例中即“gil”。文件的组别为“develop”, 对应模式为“r-x”, 而其他用户的模式为“--x”。这意味着用户 gil 对该文件拥有所有权限, 甚至有权对其进行修改或删除; 组别“develop”的其他成员可以对该文件进行读和执行 (如果该文件是一个可执行脚本或程序), 但不能对其进行写操作; 而其他用户则只拥有对该文件的执行权。用来设置这些访问权限的 `chmod` 命令为:

```
chmod 751 spellcheck
```


其中，7 对应于二进制数 111，即准许所有访问权限；5 对应于二进制数 101，即准许读和执行权限。

如果我们希望准许组别 “develop” 对有关文件进行操作（如修改等），我们一般会使用另一个命令，即 `chgrp`，寓意 “改变组别”。进一步说，我们可以输入：

```
chgrp develop spellcheck
```

在 Linux 以及当前其他的类 UNIX 的操作系统中，已经对这种神秘秘的 `chmod` 命令用法进行了增强，提供了更多的符号参数支持。例如，如下的命令：

```
chmod g+w spellcheck
```

用来为相应文件所指定的组别的访问权限中增加写权限。

6.2.2 文件控制块

由于有多个用户和运行着多个进程，所以可能有两个或多个用户正在处理一些相同的文件，不过他们可能正在处理文件的不同部分，如图 6-4 所示。文件控制块相关结构分为两部分，以支持这种应用场景并且力争信息重复量最少。就像我们可以看到的那样，有一个系统范围的打开文件表（systemwide open file table，简称系统打开文件表），其位于内核中，包含有关文件的对于所有用户都相同的元数据，如第一个盘块的地址、文件的长度，等等。同时，每个进程还分别拥有一个进程打开文件表（per-process open file table），其每个表项包含有一个指向系统打开文件表对应表项的索引以及关于该进程使用相应文件的信息（譬如当前指针）。

[122]

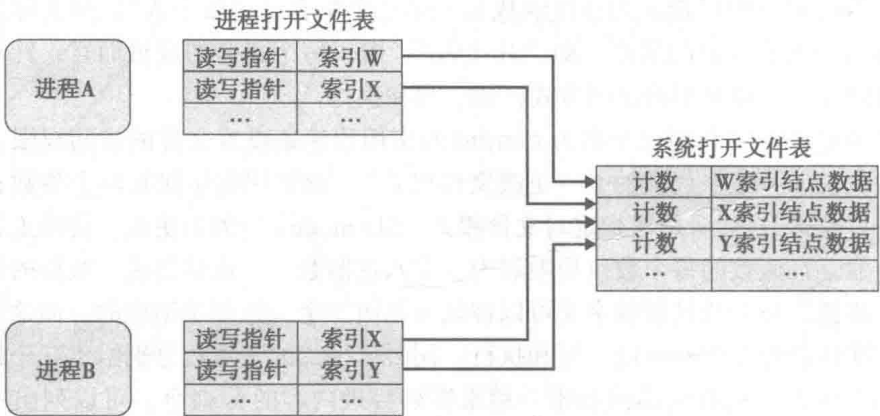


图 6-4 Linux 文件控制块

6.3 进程和线程

6.3.1 Linux 任务

我们还没有充分地讨论过线程的有关思想。在这里也是一样，因为 Linux 操作系统并不区分进程和线程，不过写作人员在写到关于 Linux 系统相关内容的时候往往还是会使用这些术语，因为它们在其他地方会经常用到。准确地说，Linux 文档采用的对应术语是任务（task）。在 UNIX 操作系统下，当一个进程（称为父进程）想要启动另一个进程（称为子进程）的时候，它首先会发出 “fork” 系统调用。该系统调用将会以父进程副本的方式

来创建子进程。(我们后面将会看到，系统可以采取多种方法来达到这个目的，而且不用实际复制对应程序的所有部分。)然而，对于 Linux 操作系统来说，相应的系统调用是 clone。就像所有的操作系统一样，Linux 操作系统为每个进程维护着若干不同的内存分段，相关内容将在后面展开更为详细的描述。需要指出的是，clone 系统调用会指定一组标志，以告知操作系统要在父进程和子进程之间共享这些内存分段中的哪些分段。有关标志如表 6-2 所示。

表 6-2 Linux 操作系统 clone 系统调用参数所用到的标志

CLONE_VM	共享内存空间
CLONE_FILES	共享文件描述符
CLONE_SIGHAND	共享信号处理程序
CLONE_VFORK	允许子进程在停止运行时向父进程发信号
CLONE_PID	共享进程标识符
CLONE_FS	共享文件系统

为了支持为其他 UNIX 系统编写的程序，Linux 操作系统还必须设法支持标准 UNIX 用于派生进程的调用。然而令人遗憾的是，Linux 操作系统提供的进程复制函数 (clone) 并没有提供相同的功能。进一步说，在执行 clone 系统调用时，用于支持任务的若干数据结构 (包括访问权限) 不会自动地实现在父任务和子任务之间的共享。为此，打算支持 POSIX 兼容性的相关库必须得自己提供此项服务。

123

6.3.2 抢占式多任务

当单个用户运行多道程序的时候，其中只会有一道程序是交互式的。在这种情况下，如果该应用程序所使用的处理器时间比公平分享份额多些，一般也不会有什么問題，因为在交互式程序中发生了一些冗长处理操作的时候，有关用户往往也不会介意其他程序是否不时地出现了暂停。然而，在多用户系统中，为一个用户运行的程序不应该“粘”在处理器上和没完没了地运行。因此，与较高版本的 Mac 操作系统一样，Linux 操作系统也是一种抢占式多任务系统。这意味着当有关操作系统开始运行一个进程的时候，它将会设置一个定时器，以便在对应进程运行时间过长而未执行阻塞式系统调用的情况下，触发中断以使操作系统进行处理。换句话说，如果定时器超时，那么当时正在运行的进程将会被放回到准备就绪即将运行的进程队列中 (也就是说，从对应进程那里抢占了处理器)。这样就可以防止出现单个进程获得对处理器的控制而阻止任何其他进程运行的现象。相关现象可能是源自于应用程序中的某个错误，而导致相应程序陷入了死循环。不过，通常情况下，往往可能只是因为有关进程需要完成好多好多的处理工作。请注意，抢占本身所消耗的资源并不是用来完成实际有用的工作——它不是在代表任何用户进程执行什么事情。尽管如此，它可以让有关用户获得更为流畅的总体响应效果，并且通常被认为是更好的处理方式，即便与非抢占式调度比较起来它的效率实际要略低一些。一般来说，除了硬实时操作系统的某些部分之外，所有现代的操作系統都采用了抢占式处理。我们在第 8 章将会更全面地讨论相关问题。

6.3.3 对称多处理

多处理系统是指那些在单个系统中运行多个处理器的系统。这种体系结构已经普遍存在

于那些没有足够处理器电源动力可用于运行全部处理负载的系统上。就关于添加一套完整的第二系统（第二系统通常必须得与第一系统同步运行）的替代方案而言，多处理不失为一种能胜任且低成本的选择。其成本降低的一项原因在于，单个系统可以共享许多价格不菲的硬部件，例如电源、主存储器和辅助存储器以及主系统总线。

图 6-5 显示了典型的多处理器系统的体系结构。这是一个简化的示意图——例如，现代系统其实拥有若干不同的总线。请注意，主存和输入/输出架构被所有处理器共享。在单个处理器系统上，我们在任一给定的时刻只能执行一道程序。但是，在具有多个处理器的系统中，则可以真正地同时运行两个或更多进程（或线程）。

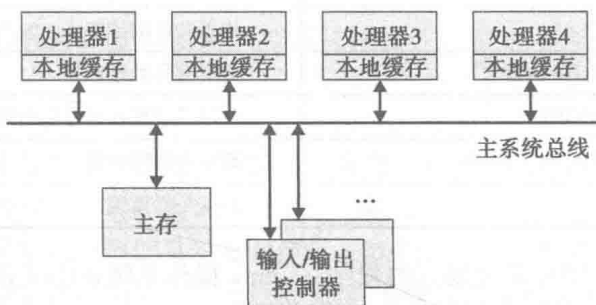


图 6-5 简化的多处理器系统体系结构

大致从 2004 年开始，集成电路设计工程师确定，把多个处理器嵌入一块芯片中而不是继续立足于让每个单独的处理器变得越来越快，其成本效益可能更为合算。相关电路被称为紧耦合多处理器（tightly coupled multiprocessor）、芯片级多处理器（Chip-level MultiProcessor, CMP）或多核处理器（MultiCore Processor, MCP）。它们甚至比以前已有的包含了多个单独的处理器芯片的多处理系统的耦合程度还要更为紧密。例如，多核处理器的电路通常还会共享单个的二级缓存（L2 cache）。这意味着，大多数像个人计算机那样大的系统将会成为多处理器系统，但是在可预见的将来，在嵌入式系统中，单处理器系统仍将十分常见。

操作系统可以采用两种不同的方法来支持多个处理器。第一种方法称为非对称多处理（asymmetric multiprocessing）。按照这种方法，操作系统只会在一个指定的处理器上运行，而其他的处理器则仅会运行应用程序。这种设计方案的优点在于，操作系统本身可以撇开和不用考虑关于同一时间在两个处理器上运行同一进程所牵涉的一些难题。尽管非常简单，但是由于有关操作系统仅仅运行在一个处理器上从而造成了性能瓶颈，所以这种方法并不经常使用。相反，大多数的现代操作系统都采用另一不同的方法，即对称多处理（Symmetric MultiProcessing, SMP）来支持多个处理器。依据这种方法，有关操作系统将会像每个其他的进程一样对待，即可以在任何处理器上运行。毋庸置疑，正在运行的程序必然会修改自己的状态（数据）。同时，对于拥有两个（或更多）处理器来运行涉及修改相同数据的相同代码的情况，显然必须得非常小心地加以处理。进一步说，必须得防止在不同处理器上运行的多个操作系统实例同时更改相同的数据结构，我们在第 9 章将会更加详细地讨论这个话题。另外，由于每个处理器可能会各自缓存相同的数据，所以相关硬件必须得做大量的工作来确保所有的缓存包含的是相同的信息。这种同步所涉及的技术具有较大的开销，从而使得大多数当前的系统往往不会扩展超越很少数量的处理器规模——64 个左右的处理器。

从 2.0 版内核开始，Linux 操作系统已经支持对称多处理体系结构。

6.4 小结

在这一章，我们讨论了一个多用户操作系统（即 Linux 操作系统）的功能和相关概念。本章相当简短，只是围绕 Linux 作为多用户操作系统而具有的附加功能进行了阐述。第 19 章才是关于 Linux 操作系统组成模块的一个更传统的案例研究。

在本章的开始，我们概要介绍了 Linux 及其演变过程的点滴历史。然后，我们简要讨论了有关多用户操作系统的功能特征。接下来，我们阐明了 Linux 操作系统中的文件支持，其后，我们还概述了 Linux 操作系统中的进程和任务的调度。

在本书的下一章，我们将会讨论分布式操作系统的一个例子——该操作系统同时运行在多个系统上，并试图让许多系统作为单一的环境呈现给用户。而随后的章节将开始深入地分析和讨论操作系统的各种各样的构成要素。

125

参考文献

Beck, M., et al., *Linux Kernel Programming*, 3rd ed. Reading, MA: Addison-Wesley, 2002.
 Bovet, D. P., and M. Cesate, *Understanding the Linux Kernel*, 2nd ed. Sebastopol, CA: O'Reilly & Associates, Inc., 2003.
 Gorman, M., *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ: Prentice Hall, 2004.

Love, R. *Linux Kernel Development*. Indianapolis, IN: Sams Publishing, 2004.
 Stevens, R., *Advanced Programming in the UNIX Environment*. Boston: Addison-Wesley, 1992.
 Stevens, R., *Unix Network Programming*. Upper Saddle River, NJ: Prentice Hall, 1990.
 Yaghmour, K., *Building Embedded Systems*. Sebastopol, CA: O'Reilly & Associates, Inc., 2003.

网上资源

<http://www.linux.org> (Linux 内核开发之家)
<http://www.kernel.org> (史上重要的内核源代码库)

<http://www.tldp.org> (Linux 文档化项目)

习题

- 6.1 为什么在 Linux 操作系统中“发行版本”很重要？
- 6.2 为什么单一 UNIX 规范对于 Linux 操作系统非常重要？
- 6.3 对于大型组织而言，其 Linux 系统的标准安装可能都不会希望使用 2.7 版。这是为什么？
- 6.4 Linux 只是操作系统的内核，并且依赖于其他群体来提供所需的实用程序，才能使其成为可用的操作系统。这是否正确？
- 6.5 Linux 是一个微内核操作系统。这是否正确？
- 6.6 现代操作系统往往用在各种各样的环境中。对于 Linux 系统而言，连接有令人难以置信的各种各样的设备和控制器，涉及各式各样的不同的文件系统、磁盘调度程序，等等，而且其中大部分并不需要任何特定的安装。像 Linux 这样的操作系统是如何避免因为在大多数情况下并不需要的那些模块而过载的？
- 6.7 为什么 Linux 操作系统的中断处理程序会划分为上半部分和下半部分？
- 6.8 简要描述 Linux 的任务复制机制 (clone) 与传统的 UNIX 进程和线程的区别。
- 6.9 Linux 是一个非抢占式多任务操作系统。这是否正确？

126

并行分布式计算、集群和网格

7.1 引言

到目前为止，我们一直讨论的都是单台机器上运行的操作系统的设计。但是，现在有许多系统是针对在许多处理器一起使用的情况下的处理而设计的。在这一章，我们将讨论在多个处理器上的计算以及我们如何管理这样的系统。对于多处理器系统，有一些常见的构型，但也有许多不常见的构型。

我们首先引入在分布式处理中会遇到的几个关键概念。然后，在推出这些概念之后，我们将会在第7.3节阐明有关并行环境中的计算和编程的一些理论。接下来，在第7.4节则会就分布式系统的常见架构加以阐述。鉴于针对这样的运行环境而设计的操作系统往往会牵涉某些在单处理情况下不会出现的特殊的问题，所以在第7.5节，我们将会就相关的操作系统问题展开讨论，具体包括如需要管理什么、对应资源管理与单处理器系统有何不同、向程序设计人员和用户呈现什么样的界面。其后，在第7.6节，我们将会引入和讨论一些适合本章主题的实际系统。最后，在第7.7节对全章内容进行归纳总结。

127

7.2 关键概念

摩尔定律认为，计算机的计算能力将逐年增强。其预言，处理器内的晶体管数量每隔18~24个月便会翻一倍。通常情况下，处理器的速度也会相应增加。而内存和磁盘的容量也会以更快的速率翻倍。摩尔定律已经成为三十多年来的一项相当准确的经验法则。在过去的几年里，处理器速度通过利用处理器芯片内部的并行性而不断提高，诸如流水线(pipelining)、处理器内多个执行单元和多核集成电路等技术已经在处理器速度的持续提升方面起到了重要作用。同时，这些技术对于程序设计人员来说都是透明的[⊖]。

然而可惜的是，有一项极限——光速，即每秒 3×10^8 米——正在快速接近。这意味着在3GHz的时钟速度下，在时钟周期之间，信号在真空中只能传送10厘米。而且时钟频率越高，时钟周期内的信号传送距离越短，这便要求提高晶体管的密度和减缩构成集成电路的硅材料的内部间距，于是牵涉制作工艺和散热等一系列难题。同时，由于处理器的跨度通常会超过一厘米，所以这又限制了处理器在一个时钟周期内可以完成的工作量。为此，尽管处理器每年都在获得更快的时钟和更快的处理速度，但是上述问题还是迫使计算机设计师另辟蹊径，推行处理器（在芯片上）的并行工作机制，同时并向程序员和用户（他们不会希望为每个新的处理器芯片重新设计和重写程序）隐藏有关实现细节。我们则希望在更高级别（并行计算或集群）上也可利用我们的计算问题中的并行性，但这要求对相关程序进行一些修改，并需对操作系统和中间件进行增强。我们将围绕这些问题以及人们如何通过操作系统和

⊖ 在这里，“透明”只是意味着一道在没有相关技术的条件下正确运行的程序，在拥有相关技术支持的情况下仍然可以正确运行。这并不意味着一个熟练的程序设计人员在需要改进性能或提供额外操作的时候可能不想利用相关功能。

其他(中间件)软件来利用这些硬件设施展开讨论。

7.3 并行处理和分布式处理

通常使用术语“并行”来指代在多处地方同时进行的工作,而且,到目前为止,我们也确实是一直使用术语“并行”来表示那种意思。进一步说,我们可以采取若干种可行的方法配置多个处理器来提供并行性。在这一节中,我们先简要描述这些方法之间的差异,后面我们将会逐个加以更为详尽地阐述。

更确切地说,我们现在描述的并行处理或并行计算是指多个处理器共享一个大的内存池以及其他资源(例如磁盘和打印机)。这种类型的计算机体系结构常常称为**多处理**(Multi-Processing, MP)体系结构。今天,大多数的多处理系统都运行在采用对称多处理架构的操作系统上,就像第6章中讨论Linux操作系统时所提到的那样。尽管多处理计算机可以拥有共享公共内存的任意数量的处理器,但是对于大多数的多处理系统而言,存在着如下一般性的指导准则:

128

- 处理器共享一个公共的内存池,并且任何处理器都可以读取或写入任何内存位置(即使相应内存位置正被另一个处理器所使用)。
- 所有处理器都具有相同的类型和速度。
- 所有其他计算机资源(磁盘、网络和打印机)在所有处理器之间共享。
- 通常只有一个操作系统副本处于运行状态,并且它了解所有的处理器和共享资源。(运行多个操作系统或让操作系统只在一个处理器上运行的情形极为罕见。)
- 程序必须专门编写或修改完善以充分利用基于多个处理器运行的模式。
- 多处理系统可以拥有2个、4个或更多(通常为2的整次幂)个处理器,但是目前2个或4个处理器的多处理系统提供了最佳的成本性能比(即每美元最佳性能),甚至比单处理器系统的成本性能比还要好;而多于8个处理器的多处理系统则较为昂贵。许多机架式系统都是2个或4个处理器的多处理系统。由于硬件的原因,所以很少在单个系统中运行64个以上的处理器^①。

然而,对于分布式计算机系统来说,则具有如下公共特性:

- 不共享内存;
- 常常各自拥有自己的资源(如磁盘驱动器);
- 彼此之间通过网络进行通信;
- 有可能不使用相同的硬件;
- 在每台机器上运行单独的操作系统副本。

尽管在分布式系统中的计算机之间发送一条消息(或共享数据)可能仅仅需要几微秒,但是其通常比多处理系统上的共享存储器要慢上至少一百倍。同样,也有若干不同类型的分布式系统,且每种类型都有独特的性能特征。

集群是一种特殊类型的分布式系统。一个集群由各个单独的计算机节点组成。这些节点可以是单处理器系统,也可以是多处理系统。它们通过专用软件进行管理和保护彼此,并通过专用局域网相连接,且该专用局域网与将对应集群连接到其他资源的其他局域网是分开

① 有一些硬件构型拥有几千个处理器在共享内存。然而,有关体系结构并不是完全共享内存。相关系统被称为非均匀性内存访问(Non-Uniform Memory Access, NUMA)系统,并不属于我们在这里讨论的那种类别。

的。通常情况下，有关集群共享一个通向集群外面的连接，且一般是连接到互联网上。正常情况下，每个集群节点具有与集群中所有其他节点都相同的软件和硬件。不过，尽管不太常见，但是也有可能由非同一性的节点来构建集群。集群常常由单一组别的某个人（或一群人）来进行管理，并且对于有关集群中的每个节点而言，所有的登录用户名和密码都是相同的。这意味着一个用户可以使用单个用户名和密码在一个或多个节点上运行作业。集群中的节点通常使用存储域网络（Storage Area Network, SAN）系统和网络附属存储器（Network Attached Storage, NAS）系统来共享存储资源。此二者本质上均是作为单个联网资源而使用诸如网络文件系统（Network File System, NFS）之类的协议进行操作的磁盘池的营销术语。通常情况下，集群拥有多节点作业调度程序，其运行贯穿指定的“头节点”，支持作业、队列和工作流的管理。一个这样的调度器，即轻便型批处理系统（Portable Batch System, PBS），稍后将会在第 7.6.6 节展开讨论。

129

网格（或称网格计算机系统）由多个工作站或具有不同管理员的集群组成。故而，它们不直接共享资源，也不共享公共登录信息，甚至可能拥有完全不同的硬件和软件配置。但是，各集群的管理员已达成一致，即允许属于其他集群或计算机系统的用户的某些作业在其集群上运行。

其他常见的共享、分布式构型包括点对点（Peer-to-Peer, P2P）系统、工作站集群（Cluster of Workstation, COWS）和志愿计算系统（例如用于外星文明探寻之家 SETI@Home、物理学、生物学处理以及许多其他项目的 BOINC 系统，即伯克利开放式网络计算平台，全称为 Berkeley Open Infrastructure for Network Computing）。虽然这样的构型对于一个开发者来说通常更加难以利用，但是它们可以潜在地提供遍布全世界的几十万个节点，计算能力超级强大。

在后续的各节中，我们将会讨论这些构型在处理大型计算工作、共享数据和处理、收集结果以及监控工作进度等方面的利用和潜力。

相关理论入门

可以采用工作流（workflow）来描述要完成的工作。有关工作流用来指定需要执行的处理步骤、相关步骤的输入和输出以及各要素之间的依赖关系。通常用有向无环图（Directed Acyclic Graph, DAG）来描述整个过程，如图 7-1 所示。其中，节点 A、B、C 和 D 被表示为方框，用来代表要执行的处理任务的单位；各边则被表示为箭头，用来代表处理节点之间的依赖关系。另外，我们省略了用来描述相关处理的输入或输出。

该工作流图表示了如下的作业流程：第一步 A 必须要处理一些数据。在完成步骤 A 之后，可以运行步骤 B 或步骤 C。由于步骤 B 和步骤 C 之间不存在依赖关系，所以它们可以同时运行。在步骤 B 和步骤 C 都完成之后，则可以运行步骤 D。例如，步骤 A 读取一些数据，然后将一部分数据传递给步骤 B，将一部分数据传递给步骤 C。其后，步骤 B 和步骤 C 各自处理它们的那部分数据，并将相应的结果分别传递给负责处理它们结果的步骤 D。假设步骤 A 需要运行 10 分钟，步骤 B 需要运行 60 分钟，步骤 C 也需要运行 60 分钟，而步骤 D 需要运行 20 分钟。如果这些步骤都是在一台计算机上完成的，那么运行这些步骤所耗费的总的时间为： $(10+60+60+20)$ 分钟 = 150 分钟。但如果是在两台计算机上（忽略诸如通信之类的开销）完成的，那么整个流程在其中一个处理节点上的运行时间为 $(10+60+20)$ 分钟

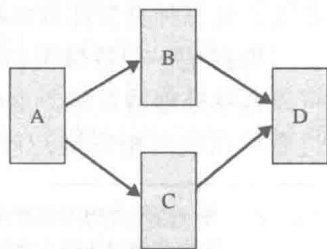


图 7-1 工作流图

(用于处理步骤 A 加步骤 B 再加步骤 D 的时间)，在另一个处理节点上的运行时间为 60 分钟 (用于处理步骤 C)，总共的处理时间也是 150 分钟。也就是说，无论哪一种情况，总的运行时间都是 150 分钟，但是对于双计算机解决方案，实际已将“挂钟”时间 (又称为观察时间) 减少到了 90 分钟，即快了一个小时。请注意，如果把步骤 D 放到第二台计算机上运行并不会对整项工作的完成有任何帮助意义，因为我们仍然必须得等待步骤 B 和步骤 C。同时，由于我们的工作流的依赖关系的特殊性，即便拥有 3 个或 4 个处理节点也不会有额外的性能提升。进一步假设我们拥有特殊的计算机可以更快地运行步骤 B 和步骤 C，那我们又能获得多少增益呢？如果我们可以将步骤 B 和步骤 C 的处理过程加快两倍，即它们每个只需要运行 30 分钟，那么我们将在 $(10+30+20)=60$ 分钟内完成整个工作流程。这通常被称为阿姆达尔定律：对工作的某一部分的提速将会使该部分更为迅捷，但对于整个工作流程而言未必会取得同样的增益效果。因此，即使在步骤 B 和步骤 C 中以 10 倍速度来进行处理，整个工作流程的速度提升亦仅为：

[130]

$$\begin{aligned} & (10+60+20) \text{ 分钟 (旧)} \\ & (10+6+20) \text{ 分钟 (新)} \qquad \qquad \qquad =2.5 \text{ 倍} \end{aligned}$$

当然，2.5 倍的速度提升还算不差，但毕竟不是 10 倍 (步骤 B 和步骤 C 的增速) 的速度增益。为此，阿姆达尔定律使得实际系统很难接近并行计算的理想效果：线性加速。线性加速意味着在 10 个节点的系统上完成的工作比在单节点系统上要快 10 倍，而在 50 个节点的系统上相关工作将会快 50 倍。有时甚至可能实现超线性加速！在 10 个节点上，有关处理速度竟然可以达到 10 倍以上！这是很不寻常的，并且通常应归功于内存中的缓存效果。也就是说，在 10 个处理器运行的时候，我们同时配备 10 倍多的缓存内存，那样将会大大加快处理速度。

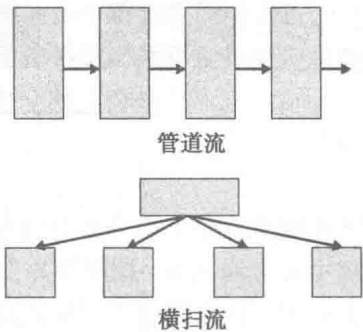


图 7-2 管道流和横扫流

工作流通常由两种结构组成，如图 7-2 所示。

管道流反映了依赖关系，而横扫流 (或称席卷流) 则可以同时并行地处理。大多数的 workflows 都是这两种模式的组合。一个非常有价值的源自直觉的结论是管道流的某一部分实际上可以允许分步处理的条件，即 (一个处理节点上) 有关管道流的某一阶段可以按每次一条记录的方式来处理数据，然后将相关结果传递到下一阶段，而后者便可以立即开始处理对应记录，与此同时，有关管道流进程的前一阶段则可以并行处理下一条记录。

关于工作流，存在如下若干令人关注的测量指标：

- 工作时间——指在所有节点上用来处理有关工作的时间总和。
- 挂钟时间 (或称时钟时间)——指处理有关工作从开始到完成所经过的时间。
- 利用率 (或称资源利用率)——指每个节点处于忙碌状态的时间百分比 (或所有节点处于忙碌状态的时间百分比的平均值)。
- 吞吐量——指每小时 (或每天) 运行的作业量或 workflow 进程数量。

[131]

7.4 分布式系统体系结构

7.4.1 执行环境概述

在各种分布式系统体系结构和单处理器、多任务系统之间存在着显著的差异。虽然每种

体系结构均允许我们并行地运行作业，但是为了利用任何一种特定的体系结构，我们必须付出的努力与利用其他架构所需付出的努力相比，有着相当大的不同。为此，本节将更为详细地讨论每一种相关可能的体系结构，以便我们能够更好地理解其间可能发生的一些问题。

正如我们在先前的讨论中所看到的，当我们利用更先进的功能时，我们往往需要小心相应的副作用以及不同功能之间的交互问题。例如，请回想一下为支持更高效的计算机资源使用而引入的并发运行若干进程的能力。但是其同时也引入了进程间通信的难题，因为我们在进程之间建立有那么多分离和保护机制。然后，我们在共享资源时还需要上锁和开锁，以避免可能出现的冲突问题。之后，我们又会担心由于锁的使用而可能产生的死锁问题。

7.4.2 对称多处理系统

由于对称多处理系统共享内存，所以处理大量数据并在各阶段之间传递数据或者共享数据表的应用程序可以从基于这种架构的运行过程中大大受益。为此，有许多常见程序（软件工具）的并行版本。或许你还记得，在对称多处理系统中，只有一个操作系统的副本在运行，且其可以在任何可用的处理器上运行。它必须管理每个处理器的进程调度以及内存分配（只有一份共享的物理内存空间）和其他资源（磁盘、终端等）。那么，如何利用对称多处理系统来并行工作呢？此类系统如图 7-3 所示。

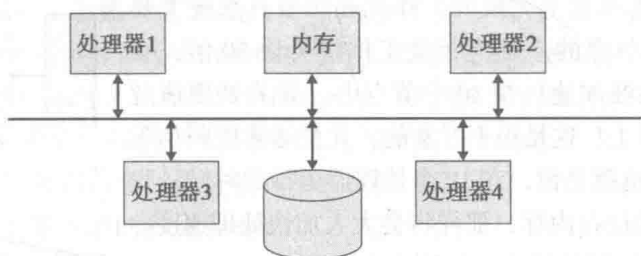


图 7-3 多处理系统

一般可以采用两种主要的技术来利用对称多处理系统的优势：多处理和多线程。（这两种技术之间的区别将会在第 8 章加以讨论。）如果这看上去有点熟悉，那是因为大多数现代的操作系统（譬如 Linux 和 Mac 操作系统）都提供有相同的设施，正如我们在前面所讨论的那样。使用对称多处理系统的关键之处在于它与传统的单处理器计算机非常相似，只是具有更大的主存容量和更多的处理器而已。

[132]

从程序设计人员的角度来看，利用多个处理器的能力可以通过简单地将整个系统划分成作为单独进程运行的许多单独程序来实现。这往往意味着运行至少与系统中的处理器一样多的进程。然而通常情况下，我们运行的进程数会比处理器数量更多一些，这是考虑到当其他进程被阻塞和处于等待状态的时候，有一些进程可以运行。一般而言，一道已经被设计和编写成创建许多进程同时运行的程序或工作流（一组程序 / 进程）可以在单处理器计算机上正常运行。但是，它们也可在对称多处理系统上正常运行，没有任何变化，只是会更快一些。（当然，在某些不太常见的情况下——例如对于几乎所有进程都被阻塞和等待输入的情况来说——常常不会有任何加速的收益。）虽然这种并行方法已经很常见，但是让多个进程共享数据还是会存在一系列困难或难题，例如前边提到的竞态条件。毋庸置疑，进程间通信和同步可以解决许多问题，但是由此所产生的开销却应当尽量通过其他方法来予以避免。如果相关工作任务可以划分为不需要很多的进程间通信和同步操作的集合（例如，划分成若干如前

所述的横扫工作流), 那么多进程模型就可以更好地发挥效用。

既然这样, 相对于其在单处理器上所做的一切, 操作系统又应当如何来管理对称多处理系统上的多处理或多线程呢? 结果证实, 并没有太大的差别。鉴于内存在一个大池中进行共享, 所以相应的内存管理与单处理器计算机上的内存管理是相同的。就处理器调度而言, 要比单处理器系统更为复杂, 因为附加的处理器必须得单独处理。不过, 就像单处理器系统一样, 时间片调度策略也常常用在对称多处理系统中, 故而使得相应设计方案的某些部分并没有很大的不同。但是, 调度器必须得考虑在哪里调度和执行进程, 因为相关任务可能被发送到不同的处理器上。然而, 这并不会比单个处理器的调度困难许多。可是, 处理器体系结构的一项最新进展则可能使有关调度变得复杂起来。请回想一下, 大多数的处理器在其芯片上都具有访问速度更快的高速缓存存储器, 其中包含有主存部分内容的副本。如果调度器随机地将进程和线程分配给各个处理器, 那么高速缓存的好处将大打折扣。是的, 整个系统仍将正常运作, 但是其运行速度将会比相应数据在合适处理器的缓存中的情况要慢得多。一旦相关进程和线程已经启动, 精致设计的对称多处理调度程序往往会试图保持进程(或来自一个进程的多个线程)自其启动以来就一直运行在同一个处理器上。这被称为**处理器偏好**(CPU preference)或**处理器亲和性**(processor affinity)。这项技术还允许程序设计人员或管理员向调度器提供建议, 以在特定的处理器上运行进程或线程。

对称多处理操作系统面临的另外一个问题是可能有操作系统的多个副本在同时运行, 且这些副本可能会尝试同时更新相同的数据。因此, 对称多处理操作系统必须得使用锁定机制来防止不同的执行副本相互干扰。这一问题将会在第9章展开更全面的讨论。

133

7.4.3 集群

与对称多处理系统相比, 集群系统的耦合性更为松散些。它们往往在每个节点上拥有基本相同的系统软件以及关于进程间共享和通信的若干选项。相应示例如图7-4所示, 其中有两组系统, 每组系统均由紧密耦合的两套系统组成(图中, 处理器编号为奇数的两套系统为一组, 而处理器编号为偶数的两套系统为另一组), 且这两组系统之间具有附加的耦合关系。另外, 每套系统都有本地内存和本地存储器。集群通常由单一的机构(例如公司或大学)进行管理。它们依赖于**中间件**(middleware), 即用来推进系统之间接口更加便利但并非操作系统组成部分的软件——其位于操作系统和应用程序之间的“中间层”上。中间件试图提供有关通过独立于所涉及的底层操作系统的方式来促进分布式处理的抽象机制。它们被认为是**平台无关的**(platform agnostic)。这便允许我们在其中, 也可以把现有的系统连接到一起, 并且让对应中间件来解决处理相关的差异问题。不过, 中间件也可以在同构的集群中加以使用。

常见的中间件软件包包括 MPI/PVM (Message Passing Interface/Parallel Virtual Machines, 消息传递接口/并行虚拟机)、CORBA (Common Object Request Broker Architecture, 通用对象请求代理体系结构)、DCOM (Distributed Component Object Model, 分布式组件对象模型)、.net 远程处理 (.net remoting) 以及 Java/RMI (Remote Method Invocation, 远程方法调用)。MPI/PVM 提供了一种与语言无关的方式来支持进程与有关集群中的其他节点上的其他进程之间进行消息、数据和参数的发送或接收, 即使相关进程是采用不同的编程语言所编写的。CORBA 是类似的, 但其允许一个对象去调用驻留在另一不同计算机上的另一个对象的方法。RMI 与 CORBA 相类似, 但仅限于 Java 语言。DCOM 则提供了一种用来调用

由微软产品所创建的远程对象上的方法的途径。它被认为是一种二进制机制，而不是一种像 CORBA 或 RMI 那样的面向语言的机制。这意味着它通过确定某种与远程对象上的分支表等同的方法来寻找其相应的目标接口。由于微软操作系统的广泛存在，所以 DCOM 也已经在大多数其他的操作系统上提供了相应的实现和支持。尽管如此，这是一种相对陈旧的机制，虽然由于其如此广泛的使用故而仍然给予了支持，但新的项目开发往往不赞成采用这种机制。进一步说，新的项目开发一般都转向采用 .net 远程处理方法。

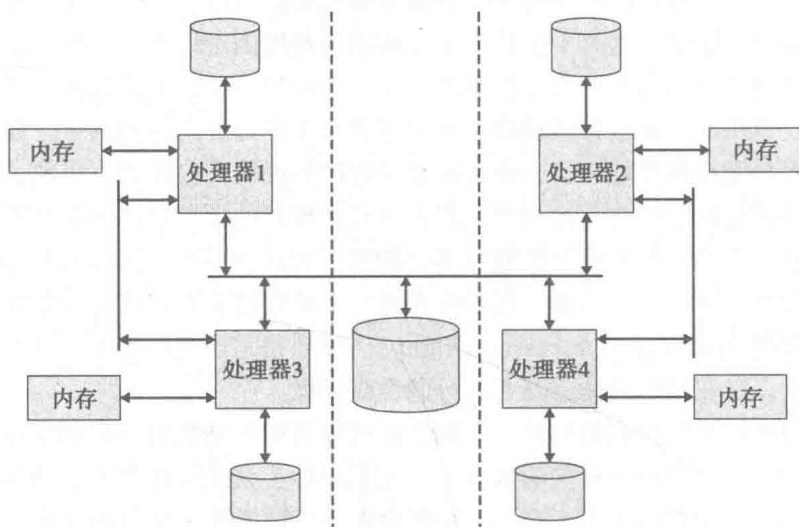


图 7-4 集群多处理系统

这些中间件软件包允许那些没有直接共享内存的进程相互之间进行信息的传递——集群的理想模式。但是，这些中间件机制实际上更适用于一般的分布式计算而非集群计算。对于为利用计算集群的并行性而设计的程序来说，则可以运用有关操作系统的其他特定的集群接口，譬如 PBS 集群调度命令，具体在后面进行讨论。

7.4.4 计算网络

网络甚至比集群的耦合性还要松散。它们是各个节点或由不同机构管理的集群的松散聚集。网络计算的主要优点在于，每个节点都可以是廉价的计算机，而通过将它们组合成网络，就能够以较低的成本产生类似于多处理器的超级计算机的计算能力。相对于构建少量的单一用途的超级计算机的较高的成本，生产日常普通的硬件商品还是非常经济的，所以其低成本优势特别明显。同时，其最大的缺点是相关节点不具备高速、低延迟的相互连接机制。因此，这种处理方案最适用于那些许多并行计算独立进行的应用程序。

通常情况下，网络中的节点不会共享用户登录和密码，而且相关节点往往具有不同的配置。尽管如此，它们一般都会运行相同的操作系统。鉴于系统内部的连接非常松散，所以无论是多线程还是 MPI、RMI，亦或类似的中间件机制都不能在网络系统中实现有效的任务分配、数据共享或工作进度监控。一个由工业界、学术界和其他相关各方组成的联合体共同构建了一套免费使用的 Globus 工具包（或称格洛伯斯工具包），广泛用来管理计算网络。该软件包是支持集群管理员将他们的某些资源作为网络的组成部分予以共享的一组实用程序、接口和协议。

由于相关节点是分开管理的,所以安全性是网络系统的一个大问题。出于安全的考量,对于相关作业而言,通常由**票据授予机构**(ticket granting agency)来授发“入网券”,而不是为其创建临时的用户登录名。许多不同的行政机构将会参与给定网格的管理。各管理人员一致信任的任何来源便可以成为票据授予机构。在网格中的节点之间传输数据和程序、预留本地空间以及检索结果均通过 Globus 命令和接口来完成。显然,站点(集群)之间的协调比在单个集群上更为困难,故而仅有极少数软件设计成是启用网格方式的。除了 Globus 之外, [135] 还有许多其他的系统也是为支持网格计算而设计的。

7.4.5 志愿计算

全世界的个人控制着数百万台的计算机,且这些计算机在大部分时间里都无所事事。即使是在运行所指定任务的情况下,大多数的个人计算机仍有许多未被利用的处理器周期,这往往是由于需要等待输入/输出任务完成而造成的。多年来,一些大项目一直希望能够利用在这些计算机上的那些被浪费掉的计算时间。于是,设计开发了许多独特的系统,以试图利用这种不用就会被浪费掉的计算机处理能力。这些系统需要处理好几方面的问题,包括允许个人在系统中注册他们的计算机、为那些计算机寻找合适作业、在对应计算机需要完成其他更重要任务的情况下允许暂停相关作业、把结果返回给发起者、记录和监测每个用户或用户组已经积累的“信用额度”。最后,威斯康星大学(University of Wisconsin)的 Condor 项目和伯克利的 BOINC 成功开发了通用的基础设施,可支持许多不同的项目按以上模式运作,且不需要针对每个项目从头白手起家来开发相关基础设施。尽管二者都提供了聚集许多不用就会被浪费掉的计算机资源的可能性,但是它们还是具有一些重要的区别。

大多数的志愿计算项目都是基于参数横扫流的,其中大量的数据被分解成小的数据集并被发送到志愿者的计算机上。这些计算机都运行着相同的科学程序来分析其自己的特定的数据集,然后将结果发回。每轮横扫所完成的工作量通常也就几个小时,并且开始启动时发送给志愿者的数据和完成结束时发送回服务器的结果通常都不会太大(几十万字节到几百万字节),从而使志愿机器负载不会太重。同时,如果作业由于某种原因而异常终止,也不会损失太多的处理结果。

BOINC

如果一个项目的计算工作可以被划分为合理大小的分块,并且使用数百万台志愿计算机将可能推动有关项目,那么 BOINC 基础设施将是颇具吸引力的。BOINC 提供了通用的基础设施,并允许项目提交计算应用程序以运行在数百万台当前存在着未被利用的处理器周期的用户计算机上。

由于早期的志愿计算系统所取得的成功,BOINC (Berkeley Open Infrastructure for Network Computing, 伯克利开放式网络计算平台)还创建了一套基础设施,可支持把任何软件系统分发到数百万台志愿计算机上共同完成。BOINC 由发出工作任务和接收处理结果的服务器系统组成。它可以配置成是使用任何志愿计算机或者优先选用有关软件已被安装且运行 Linux 或 Windows 操作系统的计算机。BOINC 客户端部分被发送到志愿的客户端计算机上,然后由其负责下载实际的科学应用程序。当用户向 BOINC 注册时,他们可以选择他们想要参与的科学项目以及应当把哪部分空闲周期用到每个项目上。接下来,BOINC 客户端软件将会处理其余的问题。它会安排科学应用程序何时运行,有可能是对应计算机在特定的 [136]

一段时间内的任何空闲时间，也有可能一直在后台利用空闲处理器周期运行。BOINC 客户端还将跟服务器一起检查和保持科学应用程序的版本为最新版本、处理与服务器之间的通信、发回运算结果。大多数的 BOINC 科学应用程序都提供了一套屏幕保护程序图形显示，通过图形、图表和动画来展示正在完成的工作。目前，BOINC 支持的项目包括一些粒子物理实验、气候预测、蛋白质结构、流行病学和医疗疾病项目、癌症研究以及外星文明探寻之家（SETI@Home）。2008 年年初，BOINC 在全球已拥有超过 250 万台活跃的计算机，提供了超过 800 TFLOPS（Trillion Floating-point Operations Per Second，每秒万亿次浮点运算操作）的计算能力。当然，这些数据还将继续向上攀升。

Condor

Condor 系统采用了一种不同的方法，其允许管理人员创建由空闲工作站组成的本地集群来实施分布式处理，且没有集群的限制或约束，也无须不辞辛劳地去构建集群设置硬件和软件或者组织网格。它提供了一套类似于 BOINC 的基础设施，但是每个项目管理其自己的单一项目以及一个专有节点集，且相关节点只可能由单一机构所拥有。

Condor 是威斯康星大学正在进行的项目，其允许用户将其计算机注册为可用的，并对相应计算机的性能加以描述：拥有什么类型的处理器（Pentium、PowerPC、Athlon 等）、多大的内存空间和磁盘空间、安装有什么软件库，等等。同时，想要运行一个程序或一组程序（工作流）的人应当以类似的方式对有关程序的要求加以描述。这些描述被称为 ClassAds（类似于分类广告一样的描述），并由 Condor 用来进行匹配撮合（即寻求提供者和请求者之间的最佳匹配）。Condor 允许计算机就什么时候完成有关工作进行偏好描述。例如，某台系统可能允许利用后台或者在无人操作键盘的几分钟内，运行相关任务。经过多年的发展，Condor 已经变得非常流行和普及，已经成为一个非常稳定的系统，且相关操作要求也只是安装一个简单程序到希望提供服务的计算机上就可以了。

常见问题

志愿计算系统必须能够就所使用的计算机的如下几方面的问题加以妥善处理：

- 它们是异构的，因此有关软件应当能够很快地适应异构配置。
- 它们加入和离开系统的时机不可预测。
- 它们的可用性是不规律的。
- 它们的正常的系统使用不应受到志愿计算系统的干扰。

此外，志愿计算系统必须处理一些关于结果可靠性的问题，这是因为志愿者常常是匿名的，故而不是不负责任的：

- 某些志愿计算机可能会出现故障并返回不正确的结果。
- 为了获得额外的信用，志愿计算机的速度可能会设置得太快，所以会更频繁地发生故障。
- 志愿计算机可能会故意返回不正确的结果。
- 志愿计算机可能为结果提出过分的信用要求。

关于所有这些问题的一种通用的解决方法是，让每批数据在至少两台计算机上进行处理。有关结果和信用只有在它们非常接近的情况下才会被接受或认可。所采用的另一项技术是对有关结果进行校验和计算或者执行循环冗余校验（Cyclic Redundancy Check，CRC，或称为循环冗余校验码）。这些是针对有关结果数据进行计算的数学函数，常常用来检测传输错误或篡改问题。

7.5 操作系统相关概念在对称多处理、集群和网格中的解读

在这一节，我们将讨论一些曾经在前面章节中有过描述的操作系统概念，并分析它们与那里讨论的单处理器系统相关概念的区别。在某些情况下，并行系统中的有关概念和实现几乎与单处理器系统一模一样；但在少数情况下，某些差异则比较重要、较为显著和值得注意。

7.5.1 进程同步和通信

请回想一下，进程常常与其他进程分工协作，而这通常也意味着对数据的共享。这样的任务分配以及数据的分解或共享往往需要进程之间的协调（译者注：即进程同步和通信）。即使是在相关可执行元素之间没有很多交互的简单情况下，也需要对那些可能运行在不同系统上的进程之间的有关共享数据（即使仅仅一个数字）的小部分的程序代码保持必要的小心谨慎。进一步说，我们应努力避免因两个进程试图同时更改单个数据项——称为**竞态条件**（race condition）——而引发问题。传统上，进程间通信一般采用共享存储器（shared memory，或称为共享内存）或消息队列来加以实现。而关于数据并发访问的同步问题则通过在相关进程的实际操作数据的所谓**临界区**（critical section）中采用信号量或类似的锁定机制来加以解决。以上这些机制往往建立在共享内存和特殊的处理器指令的基础之上，具体将在第9章中详细阐述。对于一些分布式体系结构而言，并没有提供这些机制，所以必须得使用其他的机制。关于相关系统是如何在分布式体系结构中实现同步和通信的问题，或许通过一个简单的例子最能够说明白和解释清楚。

7.5.2 一个例子

设想我们拥有一个很长的关于许多人的信息列表。例如，其中可能包括电话号码、姓名、电子邮件地址和诸如去年的家庭收入等一些数值。我们希望按电话号码升序方式对有关信息列表进行排序，同时计算平均收入。这是针对本章中所讨论的体系结构的一个非常理想的问题。（事实上，这个问题可能有点过于理想，因为它可以被结构化为高度并行的应用程序，并因而产生一个对于分布式计算可能不太典型的加速因子）。

138

解决这个问题的最显而易见的方法就是将有关列表分解成更小的、单独的列表。进一步说，如果我们拥有8个处理器来共同分担这项工作，那么我们就可以先让每个处理器针对八分之一的数据进行排序并计算相应的平均值，然后我们再对相关结果进行合并处理。各八分之一数据的处理过程就是前面曾有过描述的一种横扫流，而最后的合并步骤是管道流，同样是管道流的还有在有关工作流程的开始的时候针对数据的划分操作。在每个处理器对其自己的列表部分进行排序和计算均值的时候，在进程之间并没有交互。不过，在合并结果列表和计算整体均值的时候，会有数据共享。

如果我們可以在所有结果计算完成之前开始处理（合并）相关结果，那么将会更加高效。但是这却可能产生竞态条件，也就是说，一些处理器在所有处理器产生它们的第一批输出之前就开始尝试合并结果。进一步说，即使所有8个处理器是相同的类型和速度，但关于它们在同一时间完成它们工作任务的情况也将是非常不寻常的。是的，我们可以通过把更多的工作——列表中的更多的数据——交给更快的处理器来加以平衡。例如，如果一个处理器的运算速度是其他处理器运算速度的两倍，我们可以给它两倍的数据来进行处理。但是，鉴于排序的时间复杂度并非线性时间函数，所以对两倍数据列表进行排序所用时间将会比对单倍数

据列表排序所用时间的两倍还要长，因此，这种方案并不会奏效。总的来说，预测并行进程的运行时间非常重要，但通常情况下非常困难，而且不会非常精确。

7.5.3 例子复杂化的一面

现在，我们的简单示例已经变得比较复杂了——在各数据子列表排序结果产生的时候，对相关排序结果进行合并，同时计算相应数据的平均值（一些加法，可能结合乘法，还有除法）——并且，在我们可以使用一个横扫流进程的结果之前，我们需要知道它是否已经完成。在单处理器计算机上，这没有什么困难。我们可以利用共享内存来进行通信，并通过设置有关数据中的标志来指示完成，从而实现完成信号的发送和告知。

7.5.4 对称多处理的解决方案

这在对称多处理系统上是如何做到的呢？幸运的是，可以采用与在单处理器计算机上完全相同的方法来实现。因为，对称多处理架构在所有处理器之间共享内存，所以对于大部分用来在进程间通信的常用技术来说，可以采取和单处理器系统一样的处理方式。我们将会在第9章进一步讨论对称多处理操作系统中所涉及的相关问题。

7.5.5 集群的解决方案

[139]

如何在计算机集群上进行共享和锁定呢？与单处理器系统或对称多处理系统相比，这种体系结构较为困难。共享内存是不可能的（可以进行模拟，但是速度相当慢）。而消息必须通过局域网在各处理器节点之间传送。相关工作进行了划分和分发。但由于数据不是在处理器之间的内存中共享的，所以必须把它们分别发送给每个处理器节点。如果有关数据是源自于文件，那么可能存在跨节点的文件共享，可以在一定程度上最小化这种分布处理机制的影响。

鉴于集群中系统之间的通信要比对称多处理系统中的系统之间的通信缓慢许多，所以通常情况下，对于尝试对相关问题处理进行划分以放到集群架构上运行的方案而言，一般尽量确保在各个进程结束之前基本不要有进程之间的交互。正因如此，比起对称多处理系统或单处理器系统来，面向集群的工作流程的重写和重组需要更多的编程和设计工作，并且到最后，可以并行处理的工作不见得会有多少。不过，集群的可取之处在于集群中的每个处理节点的成本要低得多。

7.5.6 网络的解决方案

对于网络架构的系统来说，又如何来实现锁定和共享呢？这是最为困难的情况。在网格中的集群之间共享内存，从技术上讲就是不可能的，而且非常难以模拟。消息必须通过可能受到防火墙保护的而在节点之间或集群之间进行传送。有关节点可能相距甚远，故而具有非常高的通信延迟。同样，相关工作进行了划分和分发。但由于数据不是在处理器之间的内存中共享的，所以必须通过网络——主要是互联网，常常比较慢，不过也可能是第二代互联网，传送速度通常会稍好一些——把它们分别发送给每个集群。类似地，相应结果必须经由同样的网络传回来。即使有关数据是存储在文件中，被共享的文件仍然必须得复制到另一个集群，从而实现对应数据在该集群的节点之间的共享。

既然如此麻烦费力，为什么这样的努力还是值得做的呢？为什么我们要使用网格来进行

计算? 我们使用网格, 理由在于网格也是共享技术, 不过不是单纯地共享内存, 而是在用户之间共享整个计算机集群。进一步说, 一个研究人员甚至可以使用 50 个计算机集群, 而每个集群可能拥有 10~400 个左右的节点, 而不再仅仅限于只能使用本地集群中的几百个左右的节点。这种高级别的共享可以允许在一段较短的时间里使用成千上万个节点。鉴于一个人是在使用别人的集群, 所以使用时间不可能太长, 也许只有几千个小时。但是, 这个人也应当把自己的本地集群分享出去, 所以事情从长远来看还是基本平衡的, 即收支相抵的。于是, 网格的用户便由此形成了**虚拟组织** (virtual organization)。这些组织各成员一致同意聚集和共享资源 (如他们的集群)。此类组织是不断动态变化的。例如, 为了计算分析一个问题, 便可能会形成一个相应的虚拟组织。这个问题可能是一个诸如生物信息学的任务, 整个工作流程总共需要 100 000 个计算小时, 但是通过组织 24 个集群创建一个小网格来实施计算, 到周末即可完成。其后, 该虚拟组织便可解散, 直到遇到下一个大问题再重建新的虚拟组织。相比之下, 如果是在和该集群的节点相类似的单台计算机上来解决这一问题, 可能得需要 10 年以上的运行时间; 即便是利用如本章后面所描述的那样的典型的本地集群来加以处理, 也需要半年的时间。

对于非常大的数据集, 例如大型强子对撞机 (Large Hadron Collider, LHC; 是欧洲粒子物理研究所的大粒子加速器; CERN 是欧洲粒子物理研究所的前身, 即欧洲核子研究理事会的法语名称 Conseil Européen pour la Recherche Nucléaire 的首字母缩写) 的物理实验的输出, 相关分析工作将会耗费数百万的计算小时, 因此相应的虚拟组织将会存在相当长的一段时间。这些组织与摩尔定律之间存在一定的依赖关系。摩尔定律指出, 计算机及其计算能力将会逐年增加, 故而在以后的岁月里, 有关处理进度将会加快, 这样研究人员便可能会发现新的科学原理, 否则将永远不会找到。

140

7.5.7 文件共享技术

通常情况下, 大规模计算的用户往往会用到大量的文件。相关文件包含有原始数据取值、参数、中间结果、最终结果以及其他信息。其中某些文件非常庞大的情况并不少见, 有的文件甚至可能达到好几十亿字节 (GB)。对于集群而言, 拥有好几万亿字节 (TB) 的存储器 (在某些情况下, 容量可达到一百万亿字节) 颇为常见, 而前面提到的大型强子对撞机则需要千万亿字节 (PB) 级别的存储器。

对称多处理架构的文件共享相对要容易一些, 因为有关进程也共享文件系统。当然, 共享文件的相关进程可能需要利用锁或类似机制来进行协调。在大多数的对称多处理架构中, 存在一个 (或一些) 由操作系统管理的主文件系统。鉴于有关操作系统负责处理文件操作, 所以它可以在创建、读取、写入和执行其他文件操作的多个进程之间进行协调。

在集群中, 相同操作系统的多个实例运行在不同的处理器上, 并负责管理文件的共享。这具体可以通过创建特殊的文件共享节点来加以实现, 换句话说, 文件共享节点允许它们控制的文件由对应集群中的任何一个 (或多个) 节点进行操作访问。进一步说, 文件共享节点所支持的接口提供了与由单个节点或对称多处理系统中的本地操作系统基本相同的功能。鉴于对集群中的文件的访问操作也可能出现竞态条件问题, 所以文件共享节点常常还提供了锁定命令来锁定文件的全部或一部分, 从而实现无错误的数据共享。

网格不会共享文件的某部分, 也不允许在集群之间锁定。但网格允许复制整个文件, 并且某些网格工具可以模拟类似于集群那样的文件共享。确保多个集群中的所有节点拥有每个

共享文件的一致的、相同的视图是非常具有挑战性的，故而也是网络研究的活跃领域。在集群之间几乎相同的文件的管理甚至要更为困难，虽然有关文件发生了一点改变，但它们仍然具有相同的文件名。

7.5.8 远程服务的运用

应用程序经常需要访问远程服务。这些远程服务可能包括远程子程序、远程函数、对象上的方法或单独的进程。在并行和分布式计算领域，远程服务的主题已经流行了好几十年了。相关主题涵盖远程服务如何启动、如何远程调用有关服务、如何传递参数以及如何返回结果等方面。

在对称多处理系统上，特定进程之外的服务在大多数情况下通过远程过程调用（Remote Procedure Call, RPC）或远程方法调用（Remote Method Invocations, RMI）来进行调用。这与前面针对进程间管理所讨论的是相同的机制。在集群上运行的系统采用中间件对远程过程调用或远程方法调用进行了增强，以使相关调用与对称多处理系统或支持多道处理的单处理器系统上的相同调用保持一致。由于共享（特别是共享数据）所遭遇的困难以及安全的问题，所以网络系统面临巨大的挑战。理所当然地，大多数的集群管理员对支持与对应集群中他们允许远程访问的节点的直接联系非常警惕。另外，网络系统具有潜在的较长的网络延迟，所以网络服务往往由批量类型的非交互式服务器进行提供。新的网络服务模型，例如第 7.4.4 节中所讨论的新的 Globus 模型，就提供了万维网服务作为模型，并通过颁发证书（“入网券”）来提供安全性。

远程服务器和远程服务历史悠久，各种观点层出不穷，多种实现方式各有千秋，在接下来的许多年里，这一领域仍将充满争论，并继续保持重要地位。

7.5.9 故障处理

最后，我们要谈论的是一个不那么令人愉快的问题，也就是说，当系统出错时会发生什么？

因为更多的组件、更多的计算机以及更多的软件被聚合成一个更大的系统，所以系统出错的几率将大大增加，哪怕可能只是一些小的问题。这就是为什么对称多处理系统、集群和网络必须全部识别和处理偶发故障的原因所在。

什么可能发生故障呢？首先可以想到的就是硬件故障——磁盘坏了，可能是芯片烧了，于是计算机停止工作。这将会导致一个节点失败或不再响应，并丢失其正在进行的工作（结果）。网络故障发生的概率比节点故障还要高。可能是电缆松动，也可能是交换机或路由器因发生故障而失败。当然，网络或服务器也可能遭受拒绝服务（Denial of Service, DoS）攻击。（我们在第 16 章将会讨论此类攻击。）更为常见的是一个网络或路由器会变得超载和非常拥堵，故而会丢弃某些数据流。一般来说，网络故障往往会效仿节点故障。

但是软件也可能会导致故障。例如，可能是把错误版本的程序或错误版本的运行时库加载到了系统上。这是一种十分常见的问题。不幸的是，由软件缺陷所导致的故障，往往直到故障实际发生很久之后才能被检测到。

为此，在软件编写的时候就必须要考虑故障的发生与解决。例如，中间件可以使用超时而检查确认一个远程过程调用或其他的服务请求在合理的时间期限内得到了响应。并且，如果相应的服务在对应时间期限内没有得到响应，那么应当执行另一次调用，且有可能是针

对另一不同的服务器的。而如果针对原先请求的响应后来又出现了，那么只是把有关结果丢弃掉就可以了。

监控系统可以观察网络流量，并尝试检测故障，同时还可以查看单个节点或集群的性能以检测因为硬件或配置不当的软件而导致的故障。需要指出的是，这里牵涉权衡取舍的问题。例如，太少的监控将会导致故障很长时间不被注意和未予处理，但是，太多的监控则会给计算资源和网络带宽带来巨大的开销。

7.6 举例说明

7.6.1 在集群和网格上的科学计算

在过去的几年中，若干具有重要意义的、计算密集型的自然科学项目已经使用了大型的计算集群和计算网格。在这一节，我们将讨论一些这样的项目。伴随普通商品计算机、磁盘存储系统、高速网络设备、网络带宽以及控制工作任务和数据分配的软件的价格的不断下降，此类系统的费用最近已经达到了大多数研究团体可以承受的程度。因此，许多新项目在过去的一年才取得了成果，同时还有其他的一些项目至今尚未达到这样的里程碑阶段。下面的这些项目并非是最大的，也可能不是最具重大意义的，然而，它们是运用不同方法和技术完成计算密集型工作任务的具有代表性的例子。

142

7.6.2 人类基因组脱氧核糖核酸组装

在 20 世纪 90 年代初，J. Craig Venter 建议对大型基因组采用全基因组鸟枪组装法。（利用目前的技术不可能在非常长的脱氧核糖核酸（Deoxyribo Nucleic Acid，DNA）片段中按每次一个的方式来简单地读取每个核苷酸（nucleotide）。对于基因组组装来说，首先得把脱氧核糖核酸链撕成许多短小的片段，然后再通过测序机器以每次高达 900 个碱基的串的方式来“读取”这些片段。其中，碱基分为腺嘌呤（adenine）、鸟嘌呤（guanine）、胞嘧啶（cytosine）和胸腺嘧啶（thymine）共 4 种，通常表示为 A、G、C 和 T。基因组组装算法通过提取所有片段并将它们彼此对齐，然后检测两个短串重叠的所有位置。有关示例如图 7-5 所示，其中可以看到关于原始碱基串的短片段的几种重叠情况。有关方法已经变得非常流行，这在很大程度上是因为有了计算机集群并可将之用于组装大量的重叠片段。虽然较小的基因组已经被文特尔使用鸟枪组装法完成了测序，但是组装人类基因组需要更强大的计算资源和非常复杂的软件。进一步说，有关方法所扫描的人类基因组的碱基对的数量要略多于 30 亿，且已经被分解成超过 5000 万个重叠片段。由于分解和读取序列的化学过程不太完美，所以有关算法着眼于在对齐两端后查找近似匹配。

原始碱基串	XXXACGATCGTCGAGTCATCGTTAGCGTAXXXX
第一个样本-A	XXXACGATCGTCGAGTCATCGTXXXXXXXXXXXX
第一个样本-B	XXXXXXXXXXXXXXXXXXXXTAGCGTAXXXX
第二个样本-A	XXXACGATGXXXXXXXXXXXXXXXXXXXXXXX
第二个样本-B	XXXXXXXXXXCTCGAGTCATCGTTAGCGTAXXXX

图 7-5 基因组组装

这项工作在人类基因组组装方面所完成的处理最初需要大约 20 000 个处理器小时。但

是采用了由 40 台四处理器对称多处理系统所组成的集群之后，仅仅几天时间便做完了。这个系统当时花费了 4000 万美元，现在对于同等处理能力的系统来说，也就花费几十万美元。

公共人类基因组计划 (Human Genome Project) 所使用的主要替代方法是组装更长的序列，即由有关片段生成更长的已知序列。这种分级方法也需要大量的计算资源。为此，定制设计和开发编写了一个称为 GigAssembler 的程序，并让其运行在一个由 100 个节点所组成的 Linux 集群上。以上这两种方法，计算需求足够大到要求使用计算集群。在这样的情况下，真的没有其他合理的选择方案。

143

7.6.3 IBM 计算生物学中心和集群计算

多年来，IBM 一直积极参与并行和分布式计算，在开发超大规模计算机集群和软件基础设施以及使用相关系统的生物学应用程序等方面处于领先地位，并发挥了领导作用。Blue Gene/L (蓝色基因 /L) 是一个由 131 000 个处理器组成的集群，并且具有到达每个节点的多条网络通路。该系统由 Lawrence Livermore Labs (劳伦斯利弗莫尔实验室) 与 IBM 共同设计，主要用于科学研究。需要说明的是，在全世界最大的 500 个计算集群中，大约一半是 IBM 的计算机。蓝色基因系列的计算机，所有特大型的集群，均使用相对中等速度的处理器，并采用 Linux 的修改版本作为操作系统。

计算生物学中心 (Computational Biology Center) 拥有若干令人感兴趣的大型项目，包括生物信息学、医学信息学和功能基因组学研究。其中一个项目，即所谓 Blue Matter (蓝色物质) 的生物分子模拟器，用来模拟较长时间尺度下 (几百纳秒到几微秒) 的中等大小的系统 (10 000~100 000 个原子)。对于一微秒的模拟时间，一个包含 43 000 个原子的膜蛋白系统利用蓝色基因 /L 上的 4096 个处理器来运行，所需的挂钟时间略小于两个月。

7.6.4 志愿计算集群

多年以来，利用那些不用就可能被浪费掉的处理器周期的目标已经吸引了许多人。SETI@home (外星文明探寻之家) 是一个搜寻外星人的项目，其利用了从射电望远镜收集的多年数据，相关数据一直存储在库中，但苦于没有可用的计算资源能够对其进行分析。从计算的角度来看，“外星文明探寻之家”项目已经取得了引人注目的巨大成功。多年来超过 500 万的参与者贡献了超过 200 万年的总的计算时间。在 2008 年年初，有人估算认为，在任何给定时间，“外星文明探寻之家”项目系统中的所有计算机总共可提供每秒 370 万亿次浮点运算操作 (即每秒 370×10^{12} 次浮点运算操作) 的计算能力。作为对比，蓝色基因 /L 计算集群可以达到每秒 478.2 万亿次浮点运算操作的峰值性能，且其处理器数量约为“外星文明探寻之家”项目系统处理器数量的六分之一。但请注意，“外星文明探寻之家”项目的计算机是通过家庭网络和电话线进行连接的，其中混杂着各种各样的机器，既有比较新的，也有比较旧的，有时还在为它们的用户做着其他实际的工作。虽然尚未发现确凿的外星人的迹象，但已经有了一些值得关注的发现，并对进一步的研究提供了正当理由。在最近出版的天文学刊物上表达的一个关注是，在射电望远镜上收集并通过互联网发送的数字信号有可能把地球的互联网暴露给了外星人的病毒。虽然这可以算得上是确认了外星人的存在，但在地球上尚未检测到外星人的病毒。无论如何，“外星文明探寻之家”被认为是有史以来最大的网格 / 集群计算。

Folding@home (蛋白质折叠之家) 是一项模拟蛋白质折叠和错误折叠的计划，由斯坦福的 Pande Group (潘德小组) 创立。它实现了在 5~10 微秒范围内的蛋白质折叠模拟，这一

时间尺度比以前认为可能的时间尺度长数千倍。它是第二大志愿计算项目（仅次于“外星文明探寻之家”）。2007年9月16日，“蛋白质折叠之家”项目正式宣布达到高于每秒1千万亿次浮点运算操作的性能水平。最近，它被用来分析蛋白质错误折叠，相关工作在诸如牛海绵状脑病（Bovine Spongiform Encephalopathy, BSE），即疯牛病等疾病方面具有应用价值。

144

7.6.5 一个典型的计算机集群

在这里，我们要描述一个典型的计算机集群，其拥有98台双处理器计算机。它恰巧存在，不过仅仅是为了作为此类集群的一个典型案例，并且可以支持人们在这样的环境中使用一些示例命令。对于每个节点而言，在相应计算机内部有一个本地磁盘和两个处理器，且每台计算机的两个处理器共享2GB的内存空间。这98台计算机通过一个每秒10千兆比特传输速度的交换式以太网进行相互通信以及实现与互联网之间的通信。另外，还有若干利用廉价磁盘冗余阵列（Redundant Array of Independent Disk, RAID；相关技术将在第14章予以解释说明）技术的网络附属存储器磁盘阵列，共同构成100TB的存储空间。该集群还具有连接到防火墙的5个头节点，允许外部用户连接到集群或若干专用的数据库服务器。在防火墙外面，还有一些万维网服务器（Web server，或称为网站服务器）用来完成有关系统的一般状态和信息的处理。

每个计算机节点都运行着一个作为操作系统的单独但完全相同的Linux副本，而且每个节点安装有常用的软件，例如操作系统实用程序、高级语言编译器、库和若干科学应用程序。各个计算节点和存储设备是与互联网隔离开来的，相关访问须通过上面提到的头节点进行授权。头节点上运行着集群软件，允许用户登录到头节点并通过使用PBS（Nortable Batch System，轻便型批处理系统；现在称为TORQUE，全称为Terascale Open-source Resource and QUEUE Manager，即万亿级开源资源与队列管理器；但仍然几乎总是称为PBS）运行多个并行作业。头节点还负责执行监控以及一些其他的记账工作，但是其主要是设计用作提供在多个计算节点上运行实际 workflow 服务的门户。

7.6.6 Globus 集群的使用

集群上的Linux操作系统可以很好地支持双处理器节点以及两个处理器上的调度管理。相关操作系统并不清楚它们是集群的组成部分。集群运营管理是由运行在操作系统上的中间件而非通过修改操作系统来完成的。称为PBS的中间件调度程序是免费软件，且其下面就是Linux操作系统。虽然PBS是具有许多接口的复杂系统，但是一般而言，用户仅需知道几条命令就可以有效地对其加以使用。

首先，用户必须得告诉PBS想要什么样的处理器资源，具体可以在单独的行上指定各个参数，如下所示：

```
#PBS -M dave@mymailer.uta.edu
#PBS -l nodes=10:ppn=2
#PBS -l cput=02:00:00
#PBS -l mem=213mb
#PBS -l walltime=00:20:00
```

或者也可以把最后4行整合到一起，具体为：

```
#PBS -l
nodes=10:ppn=2,cput=2:00:00,mem=213mb,walltime=00:20:00
```


[145]

这条 PBS 命令用来申请 10 个节点，且每个节点要求有两个处理器，同时还申请 213MB 的内存。该命令请求总共两个小时的处理器运行时间以及 20 分钟的挂钟时间用来运行用户即将提交的所有工作流。而参数 M 则用来告知系统：用户到底是谁。

为了查看程序的执行结果，用户需要告诉系统正常的输出流和错误输出流应该传输到什么地方。在这里，它们被重定向到了相关文件，以便后面可以提取对应结果：

```
#PBS -o outputfile
#PBS -e errorfile
```

鉴于有关作业可能需要一段时间才能完成（几周甚至几个月，在某些情况下甚至是在大型网络系统上），为此，用户可以要求在作业开始运行时发送一份电子邮件，同时在作业终止或中止时再另外发送一份电子邮件：

```
#PBS -m bae
```

最后，有关操作系统需要知道即将运行的程序的存放位置：

```
cd /temp/my_working_dir
echo "I am running on host 'hostname'"
execute my_program
rm ./junk
exit
```

特别需要说明的是，用户请求有关操作系统运行一些程序，可能是以不同的文件作为输入数据，同时还可能要求在清除任何残存的临时文件之后再退出。请注意，用户经常会将所有这些命令放到一个 shell 脚本文件中，然后运行该脚本文件即可。

利用 PBS 提交作业的用户需要牢记，该系统是一个面向批处理的系统。而大多数现代的操作系统从根本上来讲是交互式的系统——当单击图标告诉操作系统来运行一道作业时，系统会立即尝试启动相应作业。但在批处理系统中，对应作业有可能无法立即启动运行，比如可能是因为所请求的资源当时还无法使用。因此，有关作业可能先是被放置在一个队列中然后才会被调度执行。为此，用户可以使用许多命令来管理相应的作业和队列。这里简要说明其中的几种命令：

#qalter	用于更改一道批处理作业
#qdel	用于删除一道批处理作业
#qhold	用于挂起一道批处理作业
#qmove	用于把一道批处理作业移到另一个队列上
#qrls	用于释放被挂起的批处理作业
#qrerun	用于重新运行一道批处理作业
#qselect	用于选择一个特定的作业子集
#qstat	用于显示批处理作业的状态

对于那些不习惯命令行界面的用户，还可以选择使用一个称为 XPBS 的图形化用户界面版本的 PBS。

7.6.7 门户和万维网界面

当一个应用程序在集群上投入使用之后，有可能希望将其提供给其他人员访问，无论是在一个群组内还是在更广泛的社区内部。或者用户可能只是想要一个易于使用的能够访问应用程序的接口。在过去，可能会选择创建一个窗口界面，当然目前还有许多应用程序仍然是

[146]

这样做的。不过，现在也可以让网格的工作流或应用程序是基于万维网方式的。

门户就是服务器计算机，其允许用户访问数据、应用程序、信息，并共享相应结果。对于本地门户而言，允许任何人进行登录、查看正在开展的研究、匹配教职研究人员的兴趣，以及申请一个账户。而账户持有人则可以访问本地应用程序、获取数据集、与在线用户聊天以及分享数据和观点。

7.7 小结

在本章之前，我们曾经讨论了运行在单台机器上的操作系统的设计。而现代系统经常按照许多处理器一起使用的应用场景来进行设计。为此，在本章中，我们讨论了多个处理器上的计算以及在构建和使用这些系统时所出现的一些困难。我们介绍了多处理器系统的若干常见的设计方案以及一些不太常见的设计方案。在引入和定义了一些关键概念之后，我们讨论了一点并行计算的理论以及有关计算模型和编程的问题。然后，我们讨论了分布式系统的一些常见体系结构。关于在此类环境中运行的操作系统的设计而言，往往会涉及在单处理情况下不会出现的特殊考虑，所以我们介绍了在分布式系统中的操作系统所面临的一些额外的问题。相关问题包括需要管理哪些方面、多处理器系统资源管理与单处理器系统资源管理有何不同以及向程序员和用户呈现什么样的界面等。最后，我们还讨论了一些按分布式系统设计实现的实际的应用系统，其中还就某网格中的一个典型的集群系统进行了探讨。

在本书的下一部分中，我们将针对操作系统的各个主题展开更深入地讨论。

参考文献

Dubois, M., and F. A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," *Computer*, Vol. 21, No. 2, February 1988, pp. 9-21.

Geer, D., "For Programmers, Multicore Chips Mean Multiple Challenges," *Computer*, Volume 40, Issue 9, September 2007, pp 17-19.

网上资源

<http://www.globus.org> (Globus联盟主页)
<http://www.globustoolkit.org> (用于构建网格的开源软件工具包)

<http://boinc.berkeley.edu/> (伯克利开放式网络计算平台 (BOINC) 主页——“外星文明探寻之家” (SETI) 项目, 等等)

习题

- 7.1 根据摩尔定律，计算机每时每刻都在变得越来越快。那么，为什么我们还要不嫌麻烦、费力地来构建集群系统并追求其他奇特的而且要求程序员在设计中努力利用任何可能的并行性的设计方案呢？
- 7.2 对称多处理器系统和集群（几乎）总是在每个节点上使用相同的处理器，但网格系统则可以在每个节点中使用不同的处理器。这是否正确？
- 7.3 关于集群系统中的节点的描述，下面哪一项是正确的？
 - a. 它们共享单一内存。
 - b. 它们没有本地外设。
 - c. 它们通过单独的专用局域网进行通信。
 - d. 在专用节点上运行操作系统。
 - e. 以上有关“集群系统中的节点”的说法均不正确。
- 7.4 管道流是其间并行性可被利用的工作流的示例。这是否正确？
- 7.5 关于工作流的加速，阿姆达尔定律是如何描述的？

- 7.6 如下在单处理器系统中使用的通用技术中，什么技术允许程序员在对称多处理系统上利用并行性？
- a. 内存映射文件
 - b. 多线程
 - c. 临界区
 - d. 信号量
 - e. 以上都不是
- 7.7 什么常见的硬件技术要求对称多处理调度程序为进程调度做出了一些特殊的规定？
- 7.8 在集群系统上运行的分布式应用程序为了利用并行性而经常使用的机制相对应的术语是什么？
- 7.9 如下在对称多处理系统或集群中所使用的技术中，哪些技术也可用于网格系统中的分布式处理？
- a. 多线程
 - b. 远程方法调用 (RMI)
 - c. 虚拟系统
 - d. 通用对象请求代理体系结构 (CORBA)
 - e. 以上都不是
- 7.10 在单处理器系统中，当共享内存被多个进程访问时，我们必须使用临界区来对其进行保护。为什么我们通常在集群和网格上不需要使用这样的机制？
- 7.11 对于分布式系统中的大多数故障来说，通常应当采用什么机制来减轻呢？
- 7.12 如何在多处理计算机系统进行工作任务的分配？
- 7.13 如何在集群计算系统进行工作任务的分配？
- 7.14 如何在志愿计算系统进行工作任务的分配？
- 7.15 如何在 Globus 系统中进行工作任务的分配？

处理器管理及内存管理

本书的第三~五部分和大多数操作系统教材的主要部分类似。这些部分就操作系统的各个方面进行了深入的讨论。特别地，第三部分围绕所有现代操作系统必须要处理的一些比较基础的问题进行了介绍，包括进程、线程管理以及内存管理，而这两类管理则共同构成了操作系统的两大主要部分。

本部分共有4章组成。前两章主要介绍了进程、线程以及它们是如何通信和相互影响的。进一步说，第8章给出了进程的定义，并针对进程管理及调度方面的数据结构和算法的演化展开了讨论。另外，该章还定义了线程的概念，并阐明了线程是如何使用和实现的。

当开发对操作系统具有更高要求的高性能系统时，常常有必要把对应系统分解成几个独立的部分并允许它们独立地运行。为此，第9章就我们为什么会经常设计出由多个进程或线程所组成的系统的理由进行了阐述。而这样的话，相关多个进程就需要彼此通信和相互协作。正因如此，这一章还详细讨论了进程间通信的有关机制，然后指出了此类通信过程中易犯的一些错误和所牵涉的问题，进而引入了同步的概念以及由此可能导致的死锁问题。

这一部分的最后两章则针对内存管理的相关问题展开讨论。第10章介绍了简单系统中的内存管理。从某种程度上来讲，有些内容已经成为历史，然而在现在却明摆着，伴随计算机硬件的小型化，意味着我们将继续在资源稀缺的环境条件下去探索计算机的奥秘，而这些简单的方法和技术也将在可以预见的未来继续得到应用和发挥一定的作用。

第11章则阐述了大型系统中的内存管理机制。分页和分段是内存管理演化过程中形成的两种主要技术。该章首先介绍了这两种技术的工作机理，然后阐明了有效的内存访问时间的概念以及分页或分段后的内存可能受到的影响。在此基础上，引入了地址转换后援缓冲机制（即快表）的思想，并论述了这种机制是如何缓解相关问题的。接下来，该章还解释了虚拟内存的概念，并讨论了虚拟内存管理相关的一些算法。

进程管理：概念、线程和调度

在本章中，我们将讨论进程和线程。操作系统的一项基本功能就是执行程序，而正在执行的程序称为进程。程序是静态的东西。在大多数操作系统中，程序存储在辅助存储器上的文件中。最后，往往是由用户，但有时也会是另一个进程，甚至可能是运行在另一个系统上的进程，指示着操作系统去执行程序。于是，操作系统将该程序调入主存并开始执行它。那个正在执行的实体被称为进程（process）。需要注意的是，我们可以在一个系统上同时执行同一个程序的多个副本。例如，可能启动了程序开发环境的多个副本同时执行。对应程序的每个执行副本都将是一个单独的进程。常用来描述进程的其他术语还包括作业（job）或任务（task）。

[151]

在第 8.1 节中，我们将给出进程的定义，并讨论关于“进程运行所基于的机器平台”的抽象概念。操作系统必须跟踪有关每个正在执行的进程的大量的信息，特别是当相应进程实际上并不是正在处理器上运行时。在第 8.2 节中，我们解释了操作系统用来存储这些数据的主要的进程控制结构，即进程控制块。当一个进程执行时，它将会处于各种不同的状态，如准备就绪即将运行、正在运行、等待，等等。各种类型的事件会导致进程从一种状态转换到另一种状态。第 8.3 节就讨论了进程可能所处的各种状态以及可以引起状态之间转换的事件。当系统中正在执行多个进程时，操作系统必须要决定下一个即将运行的应该是哪一个进程。第 8.4 节讨论了用来调度进程加以运行的各种算法。为了使一个复杂的应用程序能够同时完成许多事情，有时希望进程启动另外的一个进程来完成某些工作，因而，第 8.5 节诠释了一个进程如何启动另一个进程的问题。

进程间切换已被证实对操作系统及其所执行的程序的性能具有非常重要的影响。为此，又提出了另外一种机制，即允许一个单独的进程通过利用线程机制来同时完成更多的事情，第 8.6 节将就此予以阐述。在第 8.7 节中，我们讨论了在一些不同的操作系统中的线程的一些真正的实现机制，在许多操作系统提供的标准线程应用程序接口以及一些高级程序设计语言中，也支持线程的使用，因此我们在这一节中也就相关话题进行了介绍。最后，在第 8.8 节，我们对全章内容进行了归纳总结。

8.1 引言

当进程运行时，它将会改变自己的状态（state）。最显而易见的是，它将会在运行时及调用子程序、函数、方法或者循环等情况下，改变程序计数器（或指令地址寄存器）的内容。在大多数机器上，它至少还会改变（处理器的）各种寄存器、系统状态寄存器和栈指针的内容。这些数据项（以及稍后讨论的其他数据项）统称为进程状态。如果我们在一个系统上仅仅运行一个进程，那么对进程状态没有什么更多要说的。但是现在，我们通常运行的不止一个进程。我们会在许多进程之间快速地切换，以试图让硬件保持忙忙碌碌的工作状态，并对用户及时地响应。

尽管我们希望系统能够同时执行许多进程，但同时我们还希望进程间的这种切换对于进

程本身来说是透明的（即进程不需要知道自己是否或何时会被挂起从而让另外一个进程启动运行）。为此，我们正在创建一个“虚拟处理器”，也就是说，每个进程都可以像它是唯一运行的进程那样运行。因为我们要完成进程间切换的所有事项，所以，当我们停止一个进程和启动另一个进程时，我们必须保存将要停止的进程的状态，并恢复我们即将启动的进程的先前状态。（这里假定，我们开始执行的进程不是一个新的过程。）我们将会在内核中创建一个结构，以保存用来描述被停止进程的状态的信息。我们称之为进程控制块（Process Control Block, PCB）。一些操作系统则把这个结构称为进程描述符（process descriptor）。

8.2 进程描述符——进程控制块

正如刚才所描述的，当一个进程由于任何原因被操作系统停止运行时，此时的处理器的状态将被保存在该进程的进程控制块中。进程控制块中还有许多其他的信息。典型的进程控制块所含内容如图 8-1 所示。不同的操作系统会在进程控制块中保持不同的信息，但以下信息通常是共有的： [152]

- 程序名
- 进程标识符，由操作系统分配，用以标识进程的一个编号
- 父进程标识符或指向父进程的进程控制块的指针
- 指向子进程的进程控制块的链表的指针
- 指向“下一个”进程控制块的指针，可能是在某个队列中
- 资源清单
- 指向打开文件表的指针
- 处理器状态信息：
 - 指令计数器
 - 栈指针
 - 系统状态寄存器
 - 其他的系统寄存器
- 事件描述符，当进程正在等待某事件时方有效
- 进程状态信息（详见下一节内容）
- 进程所有者（用户）
- 内存管理信息
- 指向某种消息队列的指针
- 指向某种事件队列的指针

当进程确实实地正在运行时，处理器的状态信息并不会被更新，认识到这一点非常重要。只有当进程由于某种原因被停止运行时，才会保存该进程的处理器状态信息。请注意，术语“状态”被“重载”和用到了多个地方。我们一直在谈论处理器的“状态”，并声称当我们停止一个进程时，我们会将这些信息保存在进程控制块中，并称之为“处理器状态信息”。或许你已经注意到，进程控制块中还有另外一个称为“进程状态信息”的信息项。这与“处理器状态信息”是两码事，下一节将会具体介绍。 [153]

8.3 进程状态和转换

操作系统的设计人员必须就其系统的外部视图提供相应的文档，以便使编程人员清楚如

何为对应的系统编写程序，而用户也能知道应该如何运行这些程序。其中，某些需要讨论的事情可以采用若干方式来进行描述，关于一个进程可能所处的“状态”的概念就是这样的一个例子。对于一个进程来说，最显而易见的状态是其正在运行。但是，任何时候每个处理器上只能有一个进程正在运行，那么此时其他进程在做什么呢？其中一些进程已为运行做好准备和蓄势待发，而还有一些进程则正在等待发生其他某类事件，然后才能继续运行。

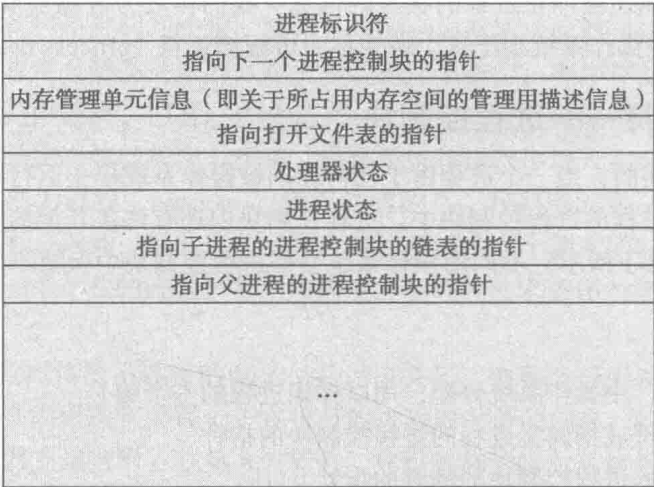


图 8-1 进程控制块

不同的设计人员（和作者）往往会采用不同的模型来解释操作系统对进程的管理。在第 2 章中，我们介绍了具有不同状态和转换标签的五状态模型，但是，去除了其中的新状态和终止状态的三状态模型（three-state model）也很常见。在这里，再次展现五状态模型，如图 8-2 所示。用状态图（state diagram）来描述这些状态是非常方便的，状态（或结点）由六边形来表示，而箭头（或转换）则对应导致从一个状态转换到另一个状态的事件。这 5 种状态是新状态、就绪状态、运行状态、等待状态和终止状态。这些状态在其他参考书中常常会有不同的名称。

新状态表示操作系统当前正准备运行但尚未准备就绪的进程。当用户告知命令处理器模块要执行一个程序时，该程序将经过标记为“0- 程序被加载”的转换，并被置为新状态。操作系统首先会为其创建一个新的进程控制块，分配一个进程标识符，并填写所有其他的进程控制块参数。然后，操作系统将为其分配内存，并把对应程序从辅助存储器读入内存，等等。特别是在多处理器系统上，我们并不希望运行在另一个处理器上的操作系统实例尝试去调度这个进程，因此直到它真正准备好可以运行之前，其一直被标记为是处于新状态。

当一个进程为运行准备妥当时，其就被置为就绪状态。这被看作“1- 进程初始化”转换。最终，对应进程将被操作系统选择为下一个要运行

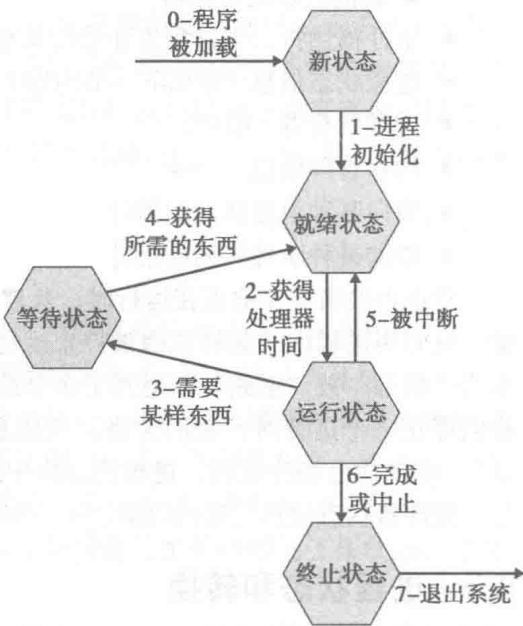


图 8-2 五状态进程模型

的进程，然后该进程将被调度分派（dispatch，即其将被置为运行状态）。相应转换由标记为“2- 获得处理器时间”的箭头来标示。当一个进程正在运行时，它可能决定在其继续运行之前需等待其他某件事情发生。一个很常见的原因是，该进程已经请求了同步输入/输出操作（譬如，普通的高级语言的读操作）并且打算一直要等到对应操作完成，因此该进程便被转换到了等待状态，有时被称为暂停（suspend）状态。相应转换被标记为“3- 需要某样东西”。我们稍后将会看到，一个进程可能等待的事件有许多不同的类型。当一个进程处于等待状态时，该进程正在等待的事件迟早可能会发生。（当然，也有可能相应事件永远不会发生，例如，一个进程正在等待可能的错误的或者关于极少使用的服务的传入请求。）作为这种转换的一个例子来讲，可能是一个进程请求的输入/输出已经完成。相应转换被标记为“4- 获得所需的東西”，并且操作系统将把对应进程置为就绪状态。这个模型的下一个转换被标记为“5- 被中断”。操作系统可以拥有好多理由来选择中断一个正在运行的进程，但并非所有的操作系统都会这么做。第一种情况是一个分时系统，其中每个进程依次被赋予少量的时间来执行。如果其中的一个进程在分配给它的时间里，没有做过什么事情使其进入等待状态，但又没有结束，那么操作系统将使其退出运行状态，并将其置为就绪状态，进而允许另外一个进程去运行。第二种情况是，一个具有高优先级别的进程一直在等待某个事件，而这时其所等待的事件发生了。如果正在运行的进程的优先级别要低于正在等待的那个进程的优先级别，那么，操作系统可能会中断较低优先级别的进程，使其返回到就绪状态，并将较高优先级别的进程置为运行状态。不过，并不是所有的操作系统都会使用优先级调度。

终止状态是为由操作系统结束的进程而专门准备的。可能有很多原因会使一个进程到达这种状态，对应转换被标记为“6- 完成或中止”。完成是显而易见的。而中止也是很清楚的，无论是进程还是操作系统检测到任何问题，相应进程都会在更多损坏发生之前被中止。但是，也有其他一些原因会使一个进程可能从运行状态转入终止状态。例如，某个进程的父进程可能判定该子进程不再需要，于是请求操作系统杀死该子进程。在大多数情况下，我们并不想弄乱这个模型，因此，我们在这张图里忽略了这些很罕见的转换。

终止状态是相当特殊的，进程不会在该状态停留很长时间，但相关进程并不是正在运行、准备就绪或等待某事发生，故而我们可以理所当然地把终止状态作为某种不同于其他状态的东西来进行讨论。操作系统将需要为对应进程做一些善后处理，例如释放进程可能已经获得但尚未返还的资源、确保文件均已关闭以及把相应的进程控制块从对应进程上拆下。在资源完全恢复之前，我们不希望选择此进程来执行，因此我们在系统工作的同时将其停留在了这种状态。

其他的操作系统文档包括更为复杂的模型。在至少一种实例^①中，设计人员所采用的模型竟然拥有9种状态及许多转换。尽管这种系统的设计人员可能觉得有必要向应用程序员解释该系统的一些特殊的方面，但是，这种级别的复杂描述在大多数的文档中都没有看到过。

值得注意的是，除了运行状态，在通常情况下，可能有许多进程同处于任何一种给定的状态下。因此，当发现“对于大多数的操作系统而言，往往都会有一种精心设计的机制用来跟踪处于任何一种其他状态的所有的进程”的时候，我们不应该感到惊奇。就绪状态将由至少一个结构组成，我们通常将其称为就绪队列（ready queue），但在技术上，我们经常会以其他方式对其进行使用，而不是采用绝对的队列操作方式。事实上，其可能是多个链表。我

① UNIX SVR4. 参 Bach, M. J., The Design of the UNIX Operating System. Englewood Cliffs, NJ: Prentice Hall, 1986. No. 1, January 1988.

们将会在下一节进一步讨论这一点。运行状态只包含一个进程，除非是多处理器系统。在那种情况下，在各个处理器上运行的进程可能会链接到一个单独的链表上，但是，仅仅把相应进程从就绪状态的链表中移除，也可能就足够了。对于等待状态而言，则可能有许多队列，鉴于处于等待状态的进程有时是以先来先服务方式进行操作的，所以将其称为队列也是合理的。对于其他的场合，我们将实施更高级的操作调度，“队列”一词实际上可能并不适用。不过，这个术语在操作系统文献中已经根深蒂固，所以我们将坚持使用这一术语，但同时应意识到，其从技术角度而言并不总是正确的。

8.4 进程调度

正如刚才所论述的，一个进程可能会由于若干原因而离开运行状态。当这种情况出现时，例如，如果进程因其时间份额用完而被中断，那么进程可能会立即进入就绪状态。如果一个进程正在等待某个事件，或许是在等待输入/输出完成吧，于是，当相应事件发生的时候，我们需要将该进程置为就绪状态，以便使其可以到达运行状态并去处理那个事件。当我们将一个进程置为就绪状态时，我们需要决定，相对于那些已经处于就绪状态的进程来说，什么时候该进程应该运行。这项决策是由所谓的短程调度器（short-term scheduler）的操作系统模块来完成的。操作系统可以基于许多方法来做出这项决定。而我们可能想要设计我们自己的操作系统，以便我们可以嵌入各种各样的短程调度程序模块，从而满足系统用户和管理员的需求。首先，我们会介绍一些调度算法，然后会讨论它们在各种情况下的一些优点和缺点。

8.4.1 先来先服务调度

最简单的方法，同时也是在历史上曾经被许多操作系统使用过的一种方法，就是仅仅使用一个普通的队列来执行先进先出的调度。这称为先来先服务（First Come, First Served, FCFS）调度算法。这种算法拥有很多优点。它很容易实现，而且能够被设计人员、用户、管理员甚至教师很好地理解。还有一点，这种算法根据定义来说应当是最公平的（也就是说，其在任何情况下都不会特殊偏爱和照顾任何一个进程，即永远不会把一个后来的进程提前到先来的进程的前面去执行）。先来先服务调度算法经常会采用这样的增强措施，即允许一个正在运行的进程执行一种可以把控制权让给处于就绪状态的下一个进程的系统调用。这被称为协作式多任务（cooperative multitasking），曾经是包括 X 版之前的 Mac 操作系统、微软 Windows 操作系统以及许多其他的操作系统在内的一代操作系统的典型做法。当然，正在运行的进程可能存在缺陷或问题，或者可能试图通过使用超过其应当所得的更多的处理器时间来使自己获得更好的用户响应，因此这并不是一种理想的解决方案。

8.4.2 优先级调度

在某些情况下，我们可能并不想使用先来先服务调度算法。首先，我们可能拥有一些进程要比其他的进程更为重要。在现代操作系统中，我们希望即时地处理击键和鼠标操作，从而使交互式用户的等待时间能够减少到最小。我们想让操纵着持有操作系统焦点的窗口的进程——假定我们正在浏览一个网站——能够得到及时的响应。我们对可能正在运行但当前并不持有焦点的其他进程——也许我们的电子邮件阅读器正在检查我们的邮件服务器，去了解我们是否收到任何的邮件——的性能不太感兴趣。我们对一些其他进程——比如说假脱

机（SPOOLING，即 Simultaneous Peripheral Operation On-Line，寓意外部设备联机同时操作）系统正在打印我们一段时间以前下载的文档——的性能更不感兴趣。在这些情况下，我们可能会使用**优先级（priority）**调度算法。在优先级调度算法中，我们将每个进程与一个优先级相关联。我们的击键和鼠标处理程序可能是最高的优先级，持有焦点的窗口的优先级位居其次，再接下来是未持有焦点的窗口的优先级，而像假脱机系统之类的后台进程的优先级则要更低一些。在大多数的操作系统中，通常都会有一个称为**空闲（idle）**进程之类的进程，该进程在没有其他进程准备就绪和可以运行的时候加以执行，并以循环结构作为基本框架。（请注意，“最高优先级”可能对应最小的数，而不是对应最大的数。这种选择可能取决于计算机的指令集，也可能只是开发人员的一个随意的决定而已。只要调度程序是兼容和一致的，最小的数来代表最高优先级是完全正常的。）

每当我们允许一些任务优先于其他任务时，往往会存在一个我们不得不担心的特殊问题。也就是说，很可能较高优先级的进程不断地推迟低优先级进程的运行，甚至导致发生极端情况即较低优先级进程从未运行过。这个问题被称为**饥饿（starvation）**问题。我们可以采用好多种方法来处理此类潜在的问题，总的来说，这些方法都会用到一种所谓的**老龄化技术（aging）**。进一步说，我们通常将监视那些被推迟的进程，并且每当我们好多次推迟一个进程时，我们将会暂时提高它的优先级。最终，它将会达到一个足够高的优先级，使其运行一次。然后，我们再将其优先级恢复为其最初的优先级。这样，即使是相当低的优先级的进程也会完成，不过，较高优先级的任务还是会被赋予大部分的处理时间。

8.4.3 保证型调度

先来先服务调度为每个进程提供了一个公平的运行机会，但是，如果一个进程会执行多次阻塞性调用，那么它比起其他的执行阻塞性调用较少的进程，则并没有真正获得公平数量的处理机时间。要做下面这样的保证是可能的，也就是说，如果有 n 个进程在同时运行，那么可以保证每个进程都获得 $1/n$ 的处理器时间。在**保证型调度（guaranteed scheduling）**中，操作系统需要监测每个进程已经使用的处理器时间总量及其分配获得的处理器时间总量。然后，计算各个进程的处理器实际利用比率，即进程实际使用处理器的时间总量与其分配获得的处理器时间总量的比值，并选择运行具有最小比率的进程。这有时被称为**公平分配调度（fair-share scheduling）**。从本质上而言，这是一种优先级调度。

157

8.4.4 最短运行时间优先调度

即便是大的面向批处理的主机也可能允许在作业之间存在优先级别。具有代表性的情况是，开发新的作业的程序员往往希望快速的作业周转，以便使他们能够完成他们的工作，而其他的作业可以通宵运行。尽管如此，一些作业会比其他作业更为重要。比如，每个人都希望工资单准时到达！当在系统中还加入分时机制的情况下，所有的交互式的基于窗口的作业通常会运行在比批处理作业更高的优先级上。使其成为现实的一种途径是采用称为**最短运行时间优先调度（Shortest Run Time First, SRTF，有时称为最短剩余时间优先调度）**或**最短作业优先调度（Shortest Job Next, SJN）**的算法。这种算法是非常容易理解的。它只是选择运行时间最短的作业作为下一个将要执行的作业。顺便说一句，这种算法将会为所有的作业产生最短可能的周转时间。

回想一下，当进程执行时，它们通常会花费一小段时间来进行计算，然后就执行输入 /

输出操作。进一步说,交互式分时作业通常在输入/输出操作之间仅运行很短的时间,而大的批处理作业在执行输入/输出操作之前则可能运行很长时间。因此,我们可以赋予交互式作业以较高优先级的一种方法是,基于进程的下一轮处理器集中使用周期且在执行输入/输出操作之前的时间量来确定其相应的优先级。然而,大多数的计算机并没有安装“具有读心术本事的程序”,因此我们通常并不知道进程的下一轮处理器集中使用周期究竟会有多长时间。尽管如此,我们可以追踪每个进程的以往的执行情况,并推测其在下一轮处理器集中使用周期的行为方式将会像以往的几轮处理器集中使用周期一样。为了能够实现这种推测,我们将使用一种按指数方式衰减的函数。在此,我们首先设定如下的变量:

T_i 表示当前进程在第 i 个时间段中实际利用处理器的时间。

E_i 表示我们推测当前进程将会在第 i 个时间段内使用处理器的时间。

同时,我们还设定参数 θ 用于调整性能,具体表示我们希望进程的上一轮的处理器实际利用时间在推测结果中所占的百分比,因此,它的取值在 0 和 1 之间。推测的其余部分则将依据我们所做的上一轮的推测结果。该推测公式给定如下:

$$E_i = (\theta * T_{i-1}) + ((1-\theta) * E_{i-1})$$

一般可把 θ 初始化设置为 0.5,这样,对这一时间段的推测结果的一半是根据上一轮的处理器实际利用时间,而另一半即 $(1-\theta)$ 则根据上一轮的推测结果。每次我们做下一轮推测的时候,以往推测值和处理器实际利用时间的影响效果将会分别减半处理。这就是为什么该函数被描述为按指数方式衰减 (exponentially decaying) 的缘故。如果我们提高 θ 的取值,那么下一轮的推测结果将更多地建立在处理器实际利用时间上。这将会使我们的推测更快地响应处理器使用的变化,但我们将会趋向于对小的波动矫枉过正。而如果我们降低 θ 的取值,那么我们将会获得相反的效果——我们将会对变化的响应非常迟钝,但不会对小的波动反应过度。

158

我们将使用这种推测方法来选择下一个要运行的进程,也就是选择那个我们推测在执行输入/输出操作之前会使用最少处理器时间的进程。于是,我们就会把那些我们认为将会耗费较多处理器时间的进程抛在后边。从这个角度来讲,最短运行时间优先调度算法是优先级调度算法的一个变种。我们只是根据我们推测的下一轮处理器集中使用周期的长度来设置进程的优先级而已。

8.4.5 高响应比优先调度

高响应比优先调度 (Highest Response Ratio Next, HRRN) 类似于最短作业优先调度,其中每道作业的优先级取决于其估计的运行时间。但是,高响应比优先调度还包含了进程用于等待的时间。如果一个进程等待的时间越长,则该进程可获得更高的优先级。采纳这种改变主要是为了减少饥饿问题发生的可能性。

$$\text{优先级} = (\text{等待时间} + \text{估计运行时间}) / \text{估计运行时间}$$

8.4.6 抢占式调度

在这些算法里,我们均假设当一个进程拥有处理器时,只要它想运行,我们将让它一直运行——通常它会因为输入/输出操作而进入等待状态。然而,如果我们正在运行优先级调度算法,并且当前运行进程的优先级非常低,我们应该怎么办呢?这里假设另一个具有较高优先级的进程一直在等待某输入/输出事件且该事件已经完成。既然我们知道这个进程比正

在运行的进程具有更高的优先级，所以我们可以把正在运行的进程停下来，并启动具有更高优先级的进程。把资源从一个进程那里夺走称为**抢占**（preemption）。在这个特例中，我们所抢占的资源就是处理器本身。

我们可以把这种抢占的理念应用到迄今为止我们学过的每一种算法里。在大多数情况下，当我们允许抢占时，我们往往会给相应的算法起一个新的名字。举例来说，如果先来先服务调度算法中我们允许实施抢占式调度，这种算法就可以称为**轮转式**（round-robin）调度算法。在这种算法里，抢占并不是根据优先级，而是根据时间额度。我们允许每个进程在不进行任何输入/输出操作的情况下运行特定的时间长度。如果当前进程的运行已经用完指定的时间长度，那么我们就把其所占用的处理器抢夺过来，调度和运行就绪队列的队首进程，并将当前进程放回到就绪队列的末尾。

如果我们把抢占式理念用到最短运行时间优先调度算法里，那么该算法就变成了**最短剩余时间优先调度算法**（shortest remaining time first algorithm）。当我们因为一个有更高优先级的进程而把处理器从当前正在运行的进程那里抢夺过来时，我们将会把被抢占进程的根据估算的运行时间而计算得到的剩余运行时间记录到其进程控制块中。于是，当我们以后重新运行该进程时，就无须重新估算其运行时间——我们只需要使用当该进程被抢占时计算得到的剩余运行时间就可以了。

在优先级调度算法里，当一个具有较高优先级的进程转入就绪状态时，我们也可以采用抢占式调度。我们并没有给这个修改后的算法一个特殊的名字，只是简单地称之为**抢占式优先级**（priority with preemption）调度算法。

159

8.4.7 多级队列调度

现代操作系统使用一种称为**多级队列**（multilevel queuing）的更复杂的调度算法。顾名思义，在这种算法里，我们将使用多个队列而非单个队列。一道新的作业将会被放置在其中的某个队列中。各个队列可以全部使用相同的调度算法，也可以使用不同的算法。如果所有算法都进行时间分片，那么每个队列可以分别拥有一个指定的不同的时间额度（即时间片大小）。这些队列将拥有不同的优先级，并且拥有最高优先级的队列会首先得到调度服务。一个必须要解决的问题是用来在队列之间共享处理器的机制，基本上有两种方法。第一种方法是，我们可以把相关机制实施成为严格的优先级调度机制。也就是说，只要在较高优先级队列中有进程，那么那些进程就应当率先运行。当然，采用这种机制，我们将不得不担心饥饿问题。另一种方法是，在队列之间共享处理器。例如，我们可能将处理器时间的 50% 专门用于第一个队列（只要该队列中有作业要运行），30% 专门用于第二个队列，20% 专门用于第三个队列。这样，由于较低优先级的队列总是会得到一些服务，所有它们将不会被饿死。

大多数现代操作系统都对多级队列调度增加了一种**反馈**（feedback）机制。最初的设想是，把任何一个新进程都看作是交互式的，因此会把它放到一个高优先级的队列里。如果相应进程在运行超过此队列允许时间片大小的过程中没有进行过任何阻塞性操作系统调用，那么操作系统可认为它并不真的是一个交互式进程，于是将其移到下一个较低优先级的队列里。这个低优先级的队列也可能拥有较长的时间片大小。需要提醒的是，上下文切换并不是生产性的工作，并且它们会由于硬件原因而使进程的执行速度暂时慢下来，我们将在后面对此加以介绍。因此，如果进程在高优先级队列的时间片内没有完成，我们可能希望在较低优先级的队列赋予它更长的时间，通常有至少三个这样的队列。于是，如果一个进程在第二个

队列的时间片内仍然没有执行任何阻塞性调用，它会被移到更低优先级的并且往往可能拥有更大的时间片的队列里。

当然，所有的进程都会有一些时间段，它们在这些时间段内会比在其他时间段做更多的计算。因此，一个大体上是交互式的进程也可能拥有一些较短的集中完成大量计算的时间段，故而被下沉到了较低优先级的队列里。为此，我们希望能拥有某种机制，允许一个进程回升到较高优先级的队列里。这可能很简单，也就是说，每当发现一个进程在当前级别队列的时间片未用完的情况下执行了一个阻塞性调用，我们就可以把它提升到一个较高优先级的队列里。然而，这可能有点反应过度了，所以我们可能会发现，有必要等到一个进程连续多次在当前级别队列的时间片内执行了阻塞性调用的情况下，再去为其执行队列提升操作。

8.4.8 最佳算法的选择

有这么多的算法，我们该如何比较它们呢？用来比较调度算法的系统性能指标有很多，其中包括：

- 吞吐量——每小时或每分钟运行的作业量。
- 平均周转时间——作业从开始到结束所经历的时间。
- 平均响应时间——从提交到开始输出所经历的时间。
- 处理器利用率——处理器运行实际作业的时间百分比（不包括进程之间切换或执行其他开销；大系统对此更感兴趣）。
- 平均等待时间——进程在就绪队列中所花费的时间。

160

前三项取决于作业行为混合情况（译者注：主要是计算与输入/输出操作的混合情况），因此很难公平和准确地进行比较。处理器利用率往往很吸引人们的注意力，并且很容易测量，但在个人计算机系统中，我们对此并不太在意。这些处理器相当便宜，我们更关心最优化察觉到的用户特性。平均等待时间是在大多数情况下最有意义的度量指标。我们希望确保大部分的计算可以在最少量的时间浪费的条件下就能够完成。平均等待时间似乎最精确地反映了这一点。

比较各种算法的平均等待时间的最简单的方法是使用所谓离散建模（discrete modeling）的方法。我们挑选进程的一个样本集以及它们的运行时间，然后我们采用手工的方式来模拟执行该样本数据的各个进程。在此基础上，我们计算那些没有运行的进程的等待时间，并比较相应结果。

首先，考虑下表的这一组进程：

进程标识符	到达时间	运行时间
1	0	20
2	2	2
3	2	2

就我们的意图而言，无所谓采用什么样的时间单位，为此，我们在这里不妨以微秒作为单位。还有，请注意，我们显示了进程 P2 和 P3 都是在 T2 时刻（这里即 2 微秒的时刻）到达的。在只有一个处理器的情况下，这两个进程是不能真正在 T2 时刻一起到达的，因为计算机一次只能做一件事情。不过，我们的时钟不是很快，所以，对于这个算法来说，我们设定它们都是在 T2 时刻到达的。对于每组数据，我们都会生成一张时间表来展示在处理器上

运行的进程。对于这组数据来讲，采用先来先服务调度算法，我们将会看到下面的时间表：



现在，让我们来计算平均等待时间。进程 P1 在 T0 时刻到达，因此它立即开始执行。进程 P2 在 T2 时刻到达，但是其并没有开始运行，而是等到 T20 时刻当进程 P1 完成时才开始执行，于是它等待了 18 个时间单位（这里即 18 微秒）。进程 P3 也在 T2 时刻到达，但同样没有开始运行，而是等到 T22 时刻当进程 P2 完成时才开始执行，于是它等待了 20 个时间单位。因此，平均等待时间就是 $(0+18+20)/3=38/3=12.67$ 。

现在假设相同的三个进程以如下表所示的稍微不同的次序到达：

进程标识符	到达时间	运行时间
1	0	2
2	2	2
3	2	20

161

相应时间表如下所示：



这一次是两个短的进程率先到达，于是进程 P1 和 P2 均没有等待，进程 P3 也仅仅等待了 2 个时间单位。因此，平均等待时间是 $(0+0+2)/3=2/3=0.67$ 。

到达时间的这一很小的差异却暴露出了先来先服务调度算法的一个主要的问题，称之为**结队效应**（convoy effect）或**“队首阻塞”**（head of line blocking）——一道短作业紧跟着一道长作业之后到达的话将不得不等待好长时间才能开始运行。这将会使运行该算法的系统呈现出平均等待时间高度变化的特征。

让我们再次来看一下第一组数据，但这次我们将为到达的作业指定优先级——最小的数对应最高的优先级：

进程标识符	到达时间	运行时间	优先级
1	0	20	4
2	2	2	2
3	2	2	1

现在，我们的时间表将是下面的这个样子：



在这种情况下，进程 P1 立即开始执行，不过，在 T2 时刻，它被后来的但却拥有最高优先级的进程 P3 抢占了处理器，于是进程 P3 立即开始执行。相应地，进程 P2 必须得等待 2 个时间单位即 T4 时刻才可开始运行，然后进程 P1 在等待 4 个时间单位之后在 T6 时刻再次启动运行。因此，平均等待时间应当是 $(4+2+0)/3=2$ 。

这个结果没有像当该组进程恰好以最优次序到达情况下的先来先服务调度算法那么好，

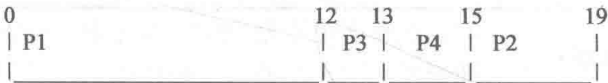
162

但却肯定比当该组进程以不好的次序到达情况下的先来先服务调度算法要好些。在这个示例中，较低优先级的作业刚好是最长的作业。当我们运行最短运行时间优先调度算法时，具有最短估计运行时间的进程应获得最高的优先级。为此，这也正是在最短运行时间优先调度算法中所发生的过程，所以，这个模拟过程也适用于那种特定类型的优先级调度算法。

接下来，让我们来看看最短运行时间优先调度算法的另外一个例子。假设我们有如下的一组进程，并且我们不允许抢占处理器：

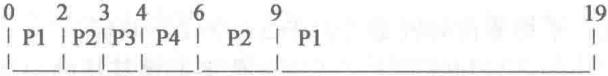
进程标识符	到达时间	运行时间
1	0	12
2	2	4
3	3	1
4	4	2

那么，对应的时间表将会是如下这个样子：



而我们的平均等待时间应当为 $(0+13+9+9)/4=7.75$ 。

如果现在假设我们允许抢占处理器，则我们的时间表应当是下面这个样子：



请注意，当前情况下这些进程本身的总的执行时间和没有抢占的情况下的总的执行时间是相同的。但是，现在的平均等待时间却是 $(7+3+0+0)/4=2.5$ 。

显然，我们更希望这个结果——但代价是什么呢？我们知道，任何事情都是有代价的。研究一下没有抢占式调度的情况，可以发现我们仅仅做了3次上下文切换，而在抢占式调度的情况下，我们则做了5次上下文切换。我们应该还记得，完成上下文切换所花费的时间并不是系统从事生产性工作的时间。也就是说，这些均属于我们花费在使平均等待时间更小（从而使许多事情，特别是高优先级的事情，看起来发生得更快些）的管理方面的开销。后面我们将会看到，上下文切换的开销甚至比只是用来保存和恢复进程处理器状态所花费的时间以及我们运行所选调度算法上所花费的时间还要大。此外，上下文切换还会在一个较短的时间段内降低硬件速度——在某些情况下会使其急剧下降。因此，我们希望执行尽可能少的上下文切换。换句话说，我们必须要认真考虑平均等待时间的典型下降与上下文切换开销（硬件系统相关）以及由此导致的进程减速之间的平衡。

163

8.4.9 长程调度器

一些操作系统还包含有所谓长程调度器（long-term scheduler）的另外一个调度程序模块。在拥有图形化用户界面的个人计算机操作系统中，通常并不存在这样的调度器。当用户点击一个图标时，与该图标相关联的进程便开始运行。在（可能除了交互式处理之外还）具有面向批处理的作业流的大型计算机系统中，系统并不会自动启动执行所有提交的作业。被提交的作业首先被放到了一个队列里，稍后才会开始执行。这就是长程调度器的工作任务，

即决定有多少道作业尝试同时执行以及具体哪些作业什么时候运行。这项决定的前一个方面是指，应当就我们希望同时运行的作业限定一个最少的道数即下限。也就是说，假如系统中要运行的作业比我们能同时运行的作业量还多，我们将会开始执行至少下限这么多道的作业。这项决定的后一个方面将和处理器利用率的水平有关。如果所有正在运行的作业大部分是那些频繁使用输入/输出操作的作业，长程调度器将会尝试去寻找一些它认为会提高处理器利用率的作业。在某种程度上，相关信息可以通过与作业一起提交的记账信息反映出来。而在其他情况下，调度器将会只是选择一道作业而已，可能是在先来先服务调度的基础上。在第11章中，我们将讨论当内存变得非常拥挤的时候这种方法可能导致的一个问题。长程调度器可以使用除了先来先服务调度算法以外的大多数的短程调度算法。鉴于长程调度器对于启动的每个进程只运行一次，所以它不需要非常迅捷，并且可以花费较多的资源来精心地选择下一道作业。

8.4.10 处理器亲和性

我们已多次提到，当一个处理器从一个进程切换到另一个进程时，会牵涉相当大的开销。由于硬件正在进行的内存高速缓存，新进程的执行将在某个时间段内显著慢下来，直到高速缓存的内容从旧的进程切换为新的进程。进一步说，我们在一个系统中可能会有一些我们认为比其他进程更为重要的进程。例如，我们的系统可能建立起来是要作为专用的数据库服务器，于是我们可能希望数据库程序具有最高的优先级。因此，在多处理器系统中，操作系统通常可能会对给定的进程维持一项所谓的**处理器亲和性**（processor affinity）。这里的亲和性是指操作系统将会用来表示对应进程在每当可能运行的情况下在某个特定处理器上运行的偏好的取值。在一些情况下，系统管理员可以指示一个特定的进程应当紧密耦合到一个特定的处理器上。而在其他情况下，操作系统将仅仅尝试让一个进程运行在其上次运行的同一处理器上。在某些操作系统中，也可能让一个处理器专门服务于一个进程，这样就只有那个进程会运行在那个处理器上。

8.5 进程创建

当一个用户向命令解释器发送信号以启动一个新的进程时，应当有一个方法能够让那个命令解释器进程启动执行该用户的相应进程。一般而言，命令解释器会使用一条标准的**访管调用**（supervisor call）来启动另一个进程，该访管调用称为**fork**。执行相应调用的进程称为**父进程**（parent process），而作为结果被启动的进程称为**子进程**（child process）。整套机制被称为“**创建（forking）一个子进程**”，有时也被称为“**产生（spawning）一个子进程**”。为此，命令解释器显然需要能够启动另一个进程，但为什么一个用户程序需要这样做呢？第一个理由是出于性能的考虑。如果一个进程拥有许多要做的任务可以同时启动，那么它就可以启动其他的进程来执行其中的某些任务，并且操作系统可以保持所有的进程都同时运行，尤其当系统拥有多个处理器时更是如此。关于为什么一个应用程序可能会被分化成若干个进程，还有其他的一些理由，我们将会就此在下一章中展开进一步的讨论。

然而，当我们让一个进程去启动另一个进程时，会产生若干问题。首先，如果父进程由于任何原因而终止，那么我们是让各个子进程继续运行呢，还是让它们也终止呢？大多数的操作系统在父进程终止的情况下，将会自动终止其各个子进程。当然也有一些操作系统例外。在大多数的现代操作系统中，我们可以有自己的选择。相应父进程已经终止的子进程称

为孤儿进程 (orphan process)。

另一个问题与资源的所有权有关。如果一个父进程拥有一个打开的文件，那么其子进程是否可以访问该文件？另一个问题则是与在子进程中所运行的程序有关。在 fork 调用的大多数情况下，子进程是其父进程在另一片内存块区中的完整的副本。请注意，父进程和子进程都将在 fork 调用之后继续执行指令。一个很有意思的问题是，每个进程如何知道哪个实例是其父进程，而哪个实例是其子进程？通常情况下，就 fork 调用的返回值而言，其对子进程将被设置为零，而对父进程则设置为正的非零数（即子进程的标识符）。以下代码是典型的 fork 系统调用的一个例子：

```
int main(void) {
    pid_t pid = fork();
    if (pid == 0) { /* 如果 pid=0, 那么我们在子进程中。*/
        do_something(from_the_child);
    }
    exit(0);
}
else if (pid > 0) { /* 如果 pid 为正数, 那么我们在父进程中, 且 pid
    为其子进程的标识符。*/
    do_something_else(from_the_parent);
}
exit(0);
}
else { /* 如果 pid 为负数, 那么出现了错误; 例如, 正在运行的进程数达
    到了最大值。*/
    fprintf(stderr, "Can't fork, error %d\n", errno);
    exit(1);
}
}
```

165

通常情况下，让父进程的另外一个实例运行并不是我们真正希望的事情。命令解释器显然属于这种情况，我们并不想要另一个命令解释器的副本。我们希望它去运行某个其他的程序。一般而言，我们真正想要的是在一个子进程中来运行另一个程序。因此，在 fork 调用之后，应执行另一个调用来把一个新的程序加载到为子进程分配的存储器空间中，这通常是一个 exec 系统调用。

当然，如果我们真正想要的是让另一个程序来运行，那么将父进程复制到另一片内存块区的最初的步骤是对许多资源的浪费。为此，一些操作系统提供了一条称为 shell 命令（或称为外壳程序命令）的不同调用。这条命令将会创建一个新进程，但不会将父进程复制到子进程空间——它会立即加载所期望的程序。一些操作系统既提供了 fork/exec 调用，又提供了 shell 命令，而其他的操作系统则仅仅提供了前者或后者中的一个。在一些系统中，高级语言库将提供一条 shell 命令，但是如果操作系统没有相应的函数调用，则对应库有可能必须得使用 fork/exec 序列来完成相关工作。

最后一个问题与子进程运行时的父进程的行动有关。类似于输入/输出的处理方式，即可以是同步的，也可以是异步的。当一个父进程创建一个子进程时，它可以选择继续执行其自身，即与子进程并行执行，或者它也可以选择等待，直到子进程完成后再继续执行。当然，也有可能父进程最初选择了继续执行，但后来需要等待直到子进程完成相应工作。为此，通常会有一个单独的 wait 系统调用，通过该系统调用，一个进程可以使自身进入等待

状态，直到子进程完成。

8.6 线程

8.6.1 什么是线程

假设我们把一个进程的逻辑地址空间作为一张图表上的纵轴。伴随时间的推移，我们不断地每隔一段时间就向右移动，并把指令计数器在该时间段到达的每一个地方做一个标记。于是，我们最后可能就会得到类似于图 8-3 那样的东西。我们可以想象，我们摊开一根线，并把它摆放在图表上，而不是用铅笔来标记空间。这是一个类比，由此便产生了短语“执行的线程”(thread of execution)。

现在，假设我们停止了这个进程，并将代表处理器的状态的所有数据保存到一张表中(我们可能称之为**线程控制块**(Thread Control Block, TCB))。然后，进一步假设我们从头开始执行这个进程。我们再次让它运行一段时间，然后让它停下来，并将处理器的状态保存在另一个线程控制块中。我们现在可以返回去，恢复所保存的对应于第一个线程的处理器状态，并继续执行第一个线程。这与运行多个进程有什么区别呢？使用多线程可能在多个方面都比使用多进程要好些。首先，我们在内存中只有一份程序和数据，所以我们会更多的内存空间用来存储其他东西。另一方面，在线程之间切换的开销比在进程之间切换的开销要小得多，因为我们只保存处理器状态。通常情况下，这意味着仅仅保存几个寄存器(包括指令指针)和指向栈空间的指针。在某些计算机上，这可以用单一的一条指令就可以完成。我们没有保存记账数据、操作系统调度信息、执行统计信息等。此外，正如我们前面所讨论的，当我们在一个进程和另一个进程之间切换处理器的时候，往往会出现硬件性能问题。我们希望尽可能避免进程上下文切换所造成的大量开销。

最后，当使用多个进程来实现一个系统的时候，这些进程是很难进行通信的。这是很自然的。操作系统设计人员克服重重困难和做了大量的努力，才使得进程彼此隔离开来，从而使一个进程不能有意或无意地改变另一进程的内存的内容。于是，许多不同的机制被创造出来以支持协作式进程之间的通信。我们在下一章中将会介绍其中的几种机制。然而，线程是没有这类问题的。根据定义，由一个进程所创建的所有线程都在同一个地址空间中运行，并共享相应的代码和数据。因此，线程间通信是鸡毛蒜皮的小事——所有线程只需访问相同的变量即可。主要的困难则在于防止各个线程同时操作相同的数据。这个问题将在下一章中进行深入讨论。

实际上，当我们开始执行第二个线程的时候，我们并不是真的在进程的一开始就启动它。回想一下，我们只是说过，各个线程共享一份程序代码及对应进程所拥有的数据。正常情况下，一个进程所做的首要事情之一就是初始化数据表。由于第一个线程已经完成了这项设置，所以我们不会想让第二个线程重做这件事情。更重要的是，第二个线程的启动不是由操作系统自己完成的。它是由正在运行的进程启动的，这样就可以让系统代表进程完成更多的工作，同时还避免了完整的进程上下文切换及由此产生的大量开销。也正是出于这个原因，线程有时被称为**轻量级进程**(lightweight process)。当一个进程正在运行时，它将会到

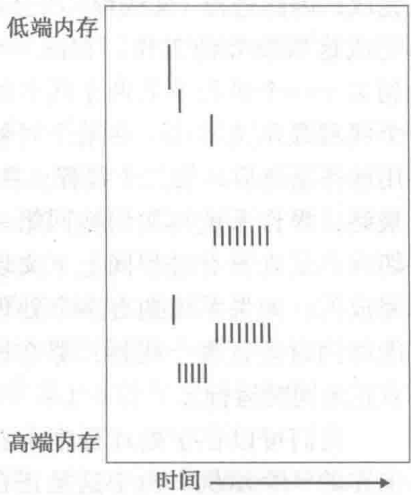


图 8-3 跟踪一个进程的指令计数器

167

达某个位置，这里有一些其他额外的工作可以与主线程（main thread）正在进行的工作并行完成，因此进程（父线程）将会启动一个子线程来完成这项额外的工作。在图 8-4 中，我们可以看到关于一个进程及其两个线程的一个例子。第一个线程显示为实线。在某个时候，第一个线程调用操作系统启动第二个线程，在图中表示为虚线。最终，操作系统再次切换回第一个线程。这两次切换都是在没有进程间上下文切换开销的情况下完成的。如果系统拥有多个处理器或一个处理器能够同时运行多个线程，那么这两个线程就可以真正地同时运行。

我们可以在字处理程序中看到关于线程如何工作的一个示例。由于这是正在写入，检查结果表明，字处理程序拥有 18 个运行的线程。有些是相当明显的，但要想出 18 个来却并非易事：

- 前台按键处理
- 显示更新
- 拼写检查器
- 语法检查器
- 重新分页
- “智能标签”识别
- 定期保存文件

另一个例子则可以在诸如万维网服务器的服务器应用程序中经常见到。一个线程即将等待传入的 HTTP 请求。对于每个传入的请求，都将启动一个新的线程，相应线程通常会执行（至少这些）步骤：

- 解析传入请求
- 查找所请求的文件
- 读取页面
- 格式化页面以方便输出
- 请求传输页面
- 结束退出

168

这个序列将每个请求的处理保持在单独的执行线程中，并且使得程序逻辑比起采用单一进程记录跟踪数百甚至数千个单独请求的状态要简单得多。

8.6.2 用户级线程与内核级线程

从历史上看，多进程的使用出现在线程的想法之前。当程序设计人员意识到在进程之间的切换使用了如此多的资源并使执行速度下降得如此厉害的时候，他们开始建立起线程的概念。然而，当时的操作系统还没有内置的线程支持机制。因此，最初的线程的开发是作为一组子程序库例程而完成的。当然，这意味着整个线程软件包是运行在用户模式下的，而操作系统并不知道应用程序正在试图保持多个活动的并行执行。因而，如果一个进程中的任何一

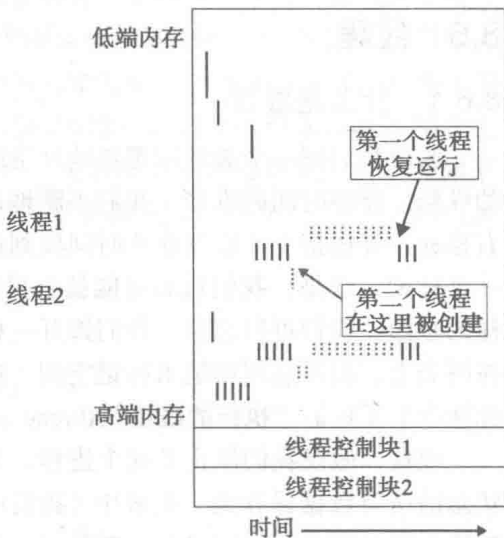


图 8-4 一个进程中共享所有代码和数据的多个线程

个线程执行了将会由于某种原因而阻塞的系统调用时，那么整个应用程序及其所有的线程都将同时被阻塞。这样的线程软件包被称为**用户级线程**（user-level thread）软件包，因为它是完全运行在用户模式下的。也正因如此，针对使用这种用户线程库的程序设计必须非常小心谨慎，从而防止因一个线程而引发整个进程的睡眠。

尽管如此，操作系统的设计人员终于做出决定：线程的想法非常好，他们应当把线程函数合并到内核中。为此，关于应用程序对线程的使用，现在的操作系统是可以察觉到的。在许多情况下，如果单个线程对操作系统进行了阻塞性调用，那么操作系统并不需要阻塞整个进程。这样的线程软件包称为**内核级线程**（kernel-level thread）。此外，由于操作系统知道各个线程的存在，所以在多处理器系统中，就可能让线程分别在独立的处理器上运行了。这是内核级线程的一项主要优势，尤其是在进入多核处理器系统时代背景——多核处理器系统将很快成为普通工作站的标准配置，而不是只有在强大的服务器上才能发现的新鲜玩意——的情况下。

一般来说，因为程序员不必想方设法去避免阻塞性调用，所以内核线程比用户线程更容易使用，这也使得其编程模型更加简单方便。例如，考虑使用线程来编写一个万维网服务程序。这类应用程序往往会采用一个循环，并在其中等待来自网络传入的 HTTP 请求命令。当一个请求传入和要求返回一个页面时，主应用程序线程便启动一个单独的线程来处理这项请求，然后返回去继续等待更多的请求。这样，正如我们前面所勾勒出来的，子线程便接受了一项非常简单的任务。进一步说，它首先会解析 HTTP 请求，然后在磁盘上查找对应页面，并通过一系列读操作把对应页面读取进来，接下来再把页面格式化为 HTTP 消息，并把结果发送出去（假设找到对应页面的前提条件下），最后结束退出。这样就无须被设计成同时处理多个请求，从而使得每个线程的设计直截了当和非常简单。另外一种方法是让主应用程序为每个输入/输出操作发出异步调用。尽管这确定无疑是可能的，但是相应模型要复杂许多，并且很难利用多处理器系统的优势。

后来还开发了另外的一种用户级线程软件包，其设计没有依赖内核级线程的支持，但却提供了在内核级线程中可以获得的编程简单性的某些优点。此软件包称为**绿色线程**（green thread）。绿色线程库捕获阻塞性系统调用，并将其转换为非阻塞性调用，在此基础上，它们处理各种用户线程的调度。这一模型允许一个程序无须修改就可运行在加载有内核级线程库的模式下，或者加载有绿色用户线程库的模式下。不过，这种方法还是有一些缺点的。首先，如果系统是多处理器系统，那么各个线程将无法利用到多处理器的优势，因为内核并不知道它们的存在。而正如我们已经提到的，处理器的发展趋势是大多数系统已经包含有多个处理器。其次，内核级线程可以采用抢占式调度方式，因此一个耗费太长时间来完成其任务的线程不可能支配整个系统，但绿色线程并没有提供这种级别的控制。

169

8.6.3 线程支持模型

当操作系统开始提供内核线程包时，应用程序设计人员并没有表现出为了使用内核线程而重写他们的应用程序的迫切希望。鉴于此，操作系统设计人员获取到了现有的用户线程库，并对它们进行了重写，以使这些用户线程库可以利用内核线程提供的相关机制。一般来说，有三种常见的方法可用于用户库例程对内核线程机制的使用，而区分它们的关键在于将用户线程映射到内核线程的方式。这三种方法具体为一对一映射模型、多对一映射模型和多对多映射模型。一对一映射相当简单。具体而言，当应用程序调用库例程来创建一个新的

线程时，库例程就调用内核线程机制来创建一个新的内核线程。图 8-5 给出了一对一线程映射模型的原理示意图。这种模型具有开发快捷简单、方便用户理解的优点。虽然其他模型似乎赋予了用户更多的控制权，但那些模型使用起来非常复杂，故而非常容易出错。大多数操作系统的供应商正在摆脱比较复杂的模型，其理由是细粒度精确控制的弊端远甚于其有利的方面。

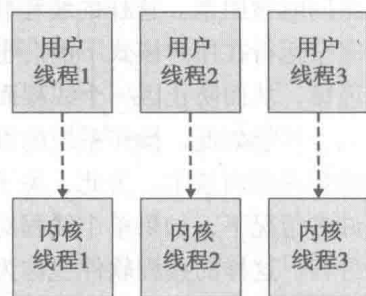


图 8-5 一对一线程映射模型

第二种映射模型称为**多对一映射模型**。回想一下，当应用程序中的任何一个线程对操作系统执行了阻塞性调用时，正在被修改的用户库是会阻塞整个进程的。在这种情况下，多对一映射模型将会做出完全相同的事情。因为仅仅创建了一个内核线程，所以所有的用户线程都将映射到这同一个内核线程上。图 8-6 给出了多对一线程映射模型的原理示意图。

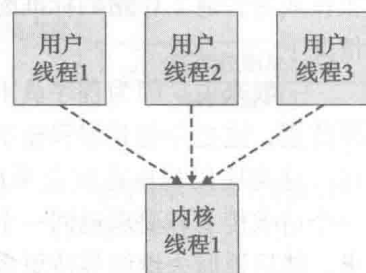


图 8-6 多对一线程映射模型

与一对一线程映射模型相比，这种模型效果差强人意，因为它没有提供内核线程的优势。它只有在操作系统没有支持内核线程的时候才会使用。在这种情况下，唯一的“内核线程”就是正在运行的进程本身。

最后一种模型称为**多对多映射模型**。在这种模型里，程序设计人员将会告知系统关于需要多少用户线程、多少内核线程以及如何映射它们的一些信息。其基本想法是，拥有一组可用的内核线程，并在需要时可以为它们动态地分配用户线程。当然，也可能是拥有多组用户线程和内核线程，并指定某些用户线程绑定到一个内核线程上。图 8-7 举例说明了多对多线程映射模型。正如前面所提到的，这种模型在理论上使用户可以对整个系统的行为实施细粒度的精确控制，但是这种模型却很难实现正确的使用，所以正渐渐丧失吸引力。特别地，现代系统已经拥有了非常庞大的内存空间和非常高速的处理能力，这样便使得用户通过采用此类复杂模型能够获取到的可感知的性能增益变得微乎其微，因而犯不上为此去遭受由此带来的程序设计难题。

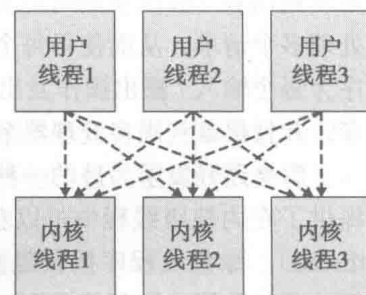


图 8-7 多对多线程映射模型

虽然线程比进程更容易创建和撤销，但是它们的创建还是会有一些开销的。因此，一些线程软件包将会在被程序第一次调用的时候创建一组线程结构，称之为**线程池（thread pool）**。当父线程要求创建一个新线程时，就从该线程池中取出一个结构，初始化为特定线程，然后予以使用。而当对应线程结束退出时，相应结构便被返回到线程池中。

线程存在的一个问题是，并不是所有的库例程都准备和设计成了“在没有完成一次调用前就开始了另一次调用”的多次调用模式。（这被称为“重入”（Reentrancy）。）当一个进程运行在具有多个处理器的系统上时，这便成了一个特别的问题。假设一个库例程被一个线程调用，并且该库例程在运行过程中使用了一个静态的局部变量。现在，同一进程的运行在另一个处理器上的另一个线程调用了相同的库例程，并尝试使用相同的静态局部变量。很容易看出，这里会有一个问题。一般来说，库例程应当通过总是在堆栈（stack，或简称栈）上分配局部变量就能够处理这种情况。按这种方式编写代码的库被称为**线程安全的（thread-safe）**，

而大多数现代的库都是线程安全的。

8.6.4 同时多线程

在同时多线程（Simultaneous Multi Threading, SMT）中，来自多个进程的指令可以同时单个处理器上执行。其实质是硬件创建了逻辑意义上的另一个处理器。这种处理器并不是一种全新的处理器，因为它主要是在两个逻辑处理器之间共享许多资源。术语“多线程”（multithreading）有点误导，因为执行线程可能来自于不同的进程。其最大的好处是，一个进程会尝试访问不在高速缓存中的数据，如果没有同时多线程技术，在数据未准备就绪的情况下，处理器将会被闲置起来。其他一般的好处则在于，当处理器的部分资源没有被一个进程使用的时候，这些资源可以被另一个进程所使用。

对处理器的主要增强包括增加在一个周期内从多个线程（或进程）加载指令的能力以及补充用于保存来自每个线程的数据的一组重复的寄存器。此外，还会增加一些我们还没有审视过的内存管理硬件。在大多数的机器上，都有一个称为快表（Translation Lookaside Buffer, TLB）的内存寻址高速缓存机构。这里的问题是每个快表项均必须包含用来标识相应表项对应于哪个逻辑处理器的相关数据，因为两个逻辑地址空间是不可能由硬件来进行区分的。同时多线程体系结构的最大优势往往会出现在当两个处理器都运行来自于一个进程的线程的情况下，因为它们能够更有效地共享资源。芯片设计的复杂性通常将逻辑处理器的数量限制为两个。还有，度量同时多线程技术的有效性可能会比较困难。在某些情况下，可以看到30%甚至更高的性能提升，但是在个别情况下，性能实际上会有所降低。今天最常见的同时多线程实现是英特尔的超线程（Hyper-Threading™）。

8.6.5 进程与线程

线程和进程都是向一个应用程序加入并行化特性的方法。进程是独立的实体，每个进程都包含自己的状态信息和地址空间。它们只有通过操作系统及进程间通信机制才能够彼此交流互动。在设计阶段，应用程序通常会被划分为若干进程。当从逻辑上分离重要的应用程序功能是有意义的情况下，一个控制进程便会调用其他进程。换句话说，进程是一种设计概念。

相比之下，线程则是一种不影响应用程序体系结构的编码技术。一个进程通常会包含多个线程。一个进程中的所有线程共享相同的状态和相同的存储空间，并且它们直接通过操纵共享数据来实现彼此的通信。

在往往被认为不是必须按顺序执行而是可以并行地运行的一系列任务的短期使用场合，一般来说就可以创建线程。当不再需要时，线程就可以被撤销掉。一个线程的范围局限于特定的代码模块中，因此我们可以将线程机制引入一个进程中，而不会影响到整个应用程序的设计。

8.7 实例研究

我们已经讨论了进程和线程，其间用到了一个理想模型以试图解释清楚它们的各种各样的特征。在真实的操作系统领域，没有操作系统会恰好按照我们所描述的那样运作。此外，虽然相关模型可能非常接近于实际情况，但操作系统文档所使用的术语可能与我们的模型并不相同。在本节中，我们将会介绍一些现代的系统，并说明它们与我们的理想化模型之间的

区别，同时也会捎带讨论一下它们的术语。

8.7.1 POSIX 线程

我们在前面已经解释过 POSIX 标准，其试图对分支激增、广泛扩散的 UNIX 操作系统的应用程序接口建立起统一的标准，而其中就有一项标准与线程有关。这项标准是众所周知的，并有一个特殊的名称 **Pthread**。然而，除 UNIX 之外，POSIX 库在许多操作系统上也都是可以利用的，因为已经使用这些应用程序接口的系统调用实现了大量的程序。或许你还记得，即使是 Windows NT 系列也拥有一个库，可以在源代码级别上支持一些 POSIX API 的系统调用。正是由于这种广泛的可用性，POSIX 线程便拥有了真正的生存发展环境：它们为应用程序提供了很高级别的可移植性。这项标准如此闻名，甚至于在 IBM Fortran 编译器中也有一个实现版本！^①尽管如此，请牢牢记住，POSIX 不是一个软件包，而是一项标准。每个实现者可以采用任何合适的方式自由地实现相关服务。

任何 POSIX 线程的实现都可以写成纯粹的用户线程软件包。但是，如果操作系统支持内核线程，那么 POSIX 线程软件包往往会使用一对一或多对多模型来实现。当开发一个应用程序且其基于 POSIX API 运行时，这便显示出了 POSIX 线程标准的不利的一面。如果系统将要运行在一个基于用户级线程实现的软件包上，那么任何线程中的一个阻塞性调用将会阻塞整个进程。但是，如果对应软件包支持内核级线程，那么操作系统可以同时为单个进程运行多个线程。因此，如果一个应用程序员真正想要充分利用多线程而不考虑要使用的特定的软件包，那么程序必须使用异步输入/输出操作来编写，从而避免阻塞整个进程。所以，[173]如果程序运行在使用内核级线程实现的环境中，那么是不会因为一个线程发出阻塞性调用就阻塞整个进程的，于是在使用异步调用开发程序方面所付出的努力就被白白浪费掉了。这是开发商为了获得 POSIX 的可移植性而必须要付出的代价。

在 Pthreads 标准中有超过 60 个可以用到的函数，而其中只有 22 个是与线程本身的基本机能有关的，其他三分之二与同步及进程间通信相关。我们将在下一章中讨论另外的这些主题。

8.7.2 Windows NT

我们正在讨论的操作系统，没有任何一种操作系统是精确地按照我们所描述的方式来实现线程和进程的。Windows NT 就是第一个这样的例子。Windows NT 的确实现了进程，但它却没有调度进程。进一步说，它为每个进程提供了一个线程，即使有关应用程序从来没有表示过它想使用线程。所以，Windows NT 调度的是线程而不是进程。这样，内核只需要操心一类被调度的实体，那就是线程。于是，一些信息被保存在进程控制块中，还有一些信息则被保存在线程控制块中。如果应用程序从不调用线程软件包以创建任何其他的线程，那么只会用到第一个线程。

Windows NT 采用多级反馈队列的调度机制，其具体使用了 32 个队列，如图 8-8 所示。其中，靠前的 16 个队列被认为是“实时”队列，而普通的应用程序则运行在靠后的 16 个队列中。就 Windows NT 而言，在其调度处于较低优先级队列中的一个线程之前，总是为较高优先级队列中处于就绪状态的所有线程提供调度服务。此外，Windows NT 是抢占式的。如果一个线程一直在等待某个事件而该事件业已发生，并且当前正在运行的线程的优先级比刚

① www-4.ibm.com/software/ad/fortran

刚转入就绪状态的线程的优先级低，那么正在运行的线程所占用的处理器将会被抢夺过来，而让刚刚转入就绪状态的线程调度运行。对于线程的运行，如果它们运行完一个时间片而没有执行任何阻塞性输入 / 输出操作，那么它们将被降级到下一个较低优先级。这里设想此类线程的行为更像一个后台任务而不是一个前台任务——即属于计算密集型的任务——所以我们会较低频度地运行此类线程。类似地，未用完时间片的线程最终将会被提升到更高优先级的队列中。当一个线程被创建时，它将配备上与其相关联的一个最高优先级和一个最低优先级，而且线程优先级的提升或降低不会超过其所关联的优先级限制范围。

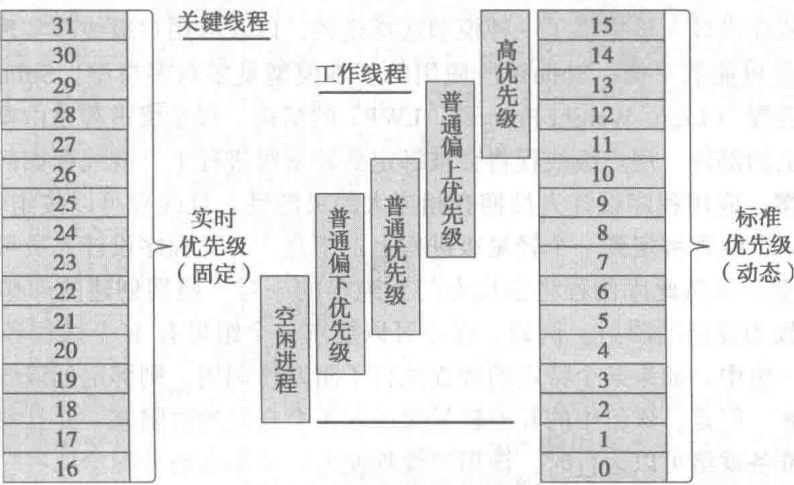


图 8-8 Windows NT 线程优先级关系

174

鉴于具有图形化用户界面的个人计算机的主要目的是使用户能够更加高效地工作，所以与图形化用户界面相关联的线程往往比运行一个应用程序的处理功能方面的线程拥有更高的优先级。此外，总有一个窗口是持有焦点的窗口。与持有焦点的窗口相关联的任何线程将会被临时提升若干优先级别，并且它们的时间片扩大为三倍。这一提升将有助于确保用户的操作得到快速的响应，同时对应线程能够完成其任务且其间不会遭受处理器被抢占的情况。

最低优先级是为称作系统空闲任务的作业预留的。只有当系统中没有其他线程准备好可以运行时，这个线程才会运行。然而在许多情况下，在个人工作站上，这个线程通常会消耗处理器 98% 左右的可利用时间。也就是说，我们的个人计算机通常在这方面比我们对它们完成我们所做的许多任务的要求快得多。这种大量的可用处理器周期正开始被诸如“外星文明探寻”(SETI[⊖])和“梅森素数大搜索”(GIMPS[⊖])此类的应用程序挖掘出来。这些程序使用志愿计算机上的空闲处理器周期来为大规模科学实验进行批量数据的处理，其间相关数据通过互联网下载和更新。这些系统类似于**网格计算**(grid computing)，即一种尝试挖掘校园环境中的桌面计算机上的未使用的处理器周期以运行某些计算密集型程序的技术。我们曾经在第 7 章讨论了这个概念，并将在第 17 章中做进一步的解释说明。

Windows NT 也可以用作服务器系统。总的来说，其服务器版本的软件代码与个人工作站版本的软件代码是相同的，主要的区别在于对系统调整参数所选定的取值。例如，其各个队列的时间片比工作站版本中的相同队列的时间片一般会大 6 倍。你应当记得，切换程序上

⊖ <http://setiathome.berkeley.edu/>
⊖ <http://www.mersenne.org/>

下文被认为是管理开销，而不是“有用”的工作，所以当我们不需要时应尽量避免上下文切换。在工作站上，整个的焦点集中在用户身上，而且无论如何都会有许多空闲的处理器周期，所以我们愿意付出额外的代价来使操作系统对用户的那些输入做出更加快速的响应。然而在服务器环境中，我们更关心针对许多服务请求的总的吞吐量，并且我们拥有较少的空闲处理器时间，所以我们会增大时间片，从而使我们在上下文切换方面花费更少的时间。

8.7.3 Solaris

[175]

Solaris 操作系统的线程支持机制曾一度是操作系统讨论的主要内容，因为其架构相当复杂，同时还为程序设计人员提供了一种模型选择途径，以获取用户级线程实现和内核级线程实现之间的最佳可能的平衡。Solaris 所使用的基本模型是多对多模型。Solaris 创建了一种所谓“轻量级进程”（Light-Weight Process, LWP）的结构。轻量级进程是由操作系统设置而运行在处理器上的部件。用户级线程将会被绑定到轻量级进程上。就线程如何被绑定到轻量级进程上的方案，应用程序设计人员拥有相当大的灵活性，且线程可以按组创建。一方面，程序可以要求一个线程绑定到一个轻量级进程上。于是，有关程序设计人员可以近似为是运用了一对一模型，虽然此库例程将会比专门为纯粹的一对一模型创建的库稍微慢一些，毕竟它还要支持较为复杂的映射。例如，程序可以要求一个组里有 M 个线程和 N 个轻量级进程。为此，在一组中，如果某个特定的线程执行了阻塞性调用，则绑定到该线程上的轻量级进程将会被阻塞。但是，该组中的其他轻量级进程不会自动地被阻塞，并且每当该组中的一个用户线程已准备就绪可以运行时，该用户线程便可以被动地分配给那些轻量级进程。此外，对于真正高性能的应用程序，还可以指定在本章前面所提到的“处理器亲和性”，因为该机制允许只有绑定到对应应用程序的轻量级进程才可运行在特定处理器（或特定的处理器组）上。

然而，从 Solaris 发行版 8 开始，这种精心设计的机制正在逐步被淘汰。Sun 公司的操作系统设计人员已确定这种机制并不值得费尽心思。也可能是反映了内存成本的持续下降吧，这种复杂的模型已逐渐被撤销。一种新的称为 T2 库的可选择的线程库被创建了，其仅仅支持一对一模型。在 Solaris 8 中仍然支持旧库，但是，对于 Solaris 发行版 9 而言，T2 模型则成为标准库，而旧的模型已被淘汰出局。Sun 公司期望，通过增强相关库的简单性，推动大多数情况下的更快操作以及更少的错误和支持问题。对于程序设计人员和系统管理员来讲，T2 模型也应当会更简单些。

8.7.4 Linux

Linux 关于进程和线程所采用的方法也不同于我们的基本模型。官方的 Linux 文献并不使用这些术语（虽然许多作者使用）。相反，他们称其为任务（task）。一个任务等同于我们一直宣称的进程。Linux 支持与大多数 UNIX 系统相同效果的 fork 系统调用，但是它使用一种称为写时复制（copy-on-write）的内存管理技术来创建子任务。这种技术允许以非常少的开销来创建子任务。写时复制技术将在第 11 章展开进一步讨论。当一个主任务通过 clone 系统调用来启动另一个任务时，Linux 便呈现出其差异性的一面。这里是 clone 系统调用的语法描述：

```
#include <sched.h>
int clone(int (*fn)(void *), void *child_stack, int flags,
         void *args);
```


第一个区别是，对于 fork 系统调用而言，父任务和子任务均会在发生该系统调用后继续执行下一条指令；而对于 clone 系统调用来说，函数名（*fn）作为参数传递给系统调用，父任务返回并在 fork 之后的下一个指令处继续执行，但是子任务将调用作为参数传入的那个函数。当该函数结束退出时，子任务便运行结束，并且该函数的值作为返回代码返回到父任务。这类似于线程调用在其他操作系统中的工作方式。

176

与 clone 系统调用的另一个主要区别则与父任务和子任务之间的共享信息有关。通常情况下，在单个任务（或进程）中运行的所有线程均会共享代码段、数据段以及其他资源（例如打开的文件），但是每个线程都有自己的线程控制块用来保存处理器状态和其自己的堆栈（可能是两个堆栈，即一个用于用户模式下的堆栈和一个用于内核模式的堆栈）。在 Linux 平台下，当一个任务克隆一个子任务时，前者将会提供一个位掩码来指定子任务与父任务对象所共享的元素。clone 系统调用可利用的一些标志包括：

- CLONE_VM——共享整个内存空间
- CLONE_FILES——共享文件描述符
- CLONE_SIGHAND——共享信号处理程序
- CLONE_PID——共享进程标识符
- CLONE_FS——共享文件系统

为解释清楚这些标志如何使用及其可能导致的不同结果，这里举例加以说明：如果子任务和父任务不共享相同的文件系统，那么当子任务执行 chdir 调用和改变当前工作目录时，则父任务的当前目录将不会改变。而如果两个任务共享相同的文件系统，则两个任务都将看到当前目录的这种变化。clone 系统调用可用来创建一个与大多数操作系统中的新进程等价的新任务。就这项操作的执行结果而言，其只不过是没在父任务和子任务之间共享任何东西而已。但是，启动等同于在大多数操作系统中的线程的一个任务牵涉共享除了进程标识符之外的所有内容。

```
clone (CLONE_VM| CLONE_FS| CLONE_FILES| CLONE_SIGHAND, 0);
```

在执行 clone 系统调用之前，父进程将为子任务分配栈空间。父进程将向 clone 系统调用传递一个指针（* child_stack）以指向为子任务所设置的堆栈空间。父进程必须确定子进程即将执行的操作需要多大的空间。一般情况下，子任务的堆栈空间将设置为与父进程的堆栈空间相同的大小。clone 系统调用的最后一个参数则是一个指针，用来指向传递给子进程即将执行的函数的参数。

8.7.5 Java

Java 程序设计语言和运行时环境是线程的一个很有趣的例子，因为 Java 是一种语言而不是一种操作系统。Java 是在语言级别上支持线程，而不是像其他编程语言那样通过子例程调用来支持线程。当然，Java 在许多不同的操作系统上都有实现。最初，Java 线程也有与 POSIX 线程相同的问题，即没有办法知道程序是以内核级线程方式还是用户级线程方式来加以执行。为此，Sun 公司为 Java 实现了两个线程库，包括一个可以在没有内核级线程支持情况下实现的“绿色”库，但该库还提供了与内核级线程所提供的相同的非阻塞模型。根据相应的操作系统，这些库可能基于内核级线程，也可能基于用户级线程。

177

8.8 小结

在本章中，我们首先定义了进程的状态及其在进程控制块的内容中被捕获的方法。然后，我们定义了各种模型来描述系统中进程的状态以及导致进程从一个状态转换到另一个状态的各种事件。接下来，我们介绍了用在操作系统进程调度中的各种算法，并讨论了如何利用确定性建模来评估这些算法。我们以进程创建的简明扼要的阐述结束了进程部分的讨论。

在此基础上，我们给出了线程的定义，并讨论了进程和线程之间的区别。然后，我们解释了用户级线程和内核级线程之间的不同。接下来，我们说明了用户级线程可能映射到内核级线程的各种方式。最后，我们介绍了现代操作系统中线程的实现，具体讨论了线程机制的几种特殊案例。

在本书的下一章，我们将讨论进程如何通信和协作，以及相关领域所涉及的一些问题。需要指出的是，这些问题并不像它们乍看起来那么简单。

参考文献

- Abbot, C., "Intervention Schedules for Real-Time Programming," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 3, May 1984, pp. 268-274.
- Bach, M. J., *The Design of the UNIX Operating System*. Englewood Cliffs, NJ: Prentice Hall, 1986.
- Brinch Hansen, P., "The Nucleus of a Multiprogramming System," *Communications of the ACM*, Vol. 13, No. 4, April 1970, pp. 238-241.
- Henry, G. J., "The Fair Share Scheduler," *Bell Systems Technical Journal*, Vol. 63, No. 8, Part 2, October 1984, pp. 1845-1857.
- Jensen, E. D., C. D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, December 3-6, 1985, pp. 112-122.
- Kay, J., and P. Lauder, "A Fair Share Scheduler," *Communications of the ACM*, Vol. 31, No. 1, January 1988, pp. 44-55.
- Liu, C. L., and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, January 1973, pp. 46-61.
- Woodside, C. M., "Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 10, October 1986, pp. 1041-1048.

网上资源

<http://web.cs.mun.ca/~paul/cs3725/material/web/notes/node19.html> (为处理器分配进程)

<http://www-4.ibm.com/software/ad/fortran>
(IBM Fortran编译器)

习题

8.1 什么是 PCB ?

- 一类有毒的化学化合物
- 进程控制块
- 程序计数器边界
- 部分完成的缓冲区
- 以上都不是

8.2 在进程的上下文中，“状态”这个词被多次用到。请就进程而言，区分这个词的两种含义。

8.3 一个进程可以处于多少种的独特的操作系统状态？

8.4 一般而言，针对就绪状态会设置多少个队列？

8.5 一般而言，针对等待状态会设置多少个队列？

8.6 为什么我们会介意一个进程调度程序是否公平？

- 8.7 按照最初的描述，最短运行时间（SRTF）进程优先调度算法是最佳的，但为什么我们不使用它呢？
- 8.8 既然先来先服务（FCFS）进程调度算法是那么公平，它又存在什么问题呢？
- 8.9 为什么具有图形化用户界面的系统通常没有长程调度器？
- 8.10 处理器亲和性的目的是什么？
- 8.11 一个进程执行什么操作来启动另一个进程呢？
- 8.12 比较进程和线程的不同之处。
- 8.13 为什么我们经常说内核级线程比用户级线程要好些？
- 8.14 用户级线程软件包是在内核级线程软件包出现之前开发完成的。当内核级线程可以使用后，用户不想丢弃或重写他们的多线程应用程序。因此，用户级线程软件包被重新编码以与内核级线程一起工作。我们谈到的用来把用户级线程映射到内核级线程的三种模型是什么？
- 8.15 当我们说一个库是“线程安全”的时候，具体是指什么意思？
- 8.16 同时多线程是指同时有多个进程创建线程。这种说法是否正确？
- 8.17 从普遍存在的意义上讲，POSIX 线程似乎是理想的。那么，POSIX 线程存在的主要缺点是什么？
- 8.18 Windows NT 进程调度有什么独特之处？
- 8.19 Linux 进程调度有什么独特之处？
- 8.20 Solaris 提供了一种精心设计的机制用于将用户级线程映射到“轻量级”进程上。为什么要这样做？
- 8.21 Java 线程有什么不寻常的地方？

进程管理进阶：进程间通信、同步和死锁

在本章，我们将继续深入讨论进程和线程的相关内容。为了保持代表有关应用程序的系统更加忙碌地运行，应用程序可以设计和划分成多个部分，我们将围绕相关技术展开讨论。当我们把应用程序分解成多个不同的部分时，这些部分之间需要相互协作，而为了做到这一点，它们便需要彼此通信。前面我们花费了相当多的时间来解释操作系统为什么要把进程独立出来以及如何实现进程间的相互隔离，现在我们则需要阐明为允许它们通信而演化形成的有关机制。

操作系统专门投入了大量的资源来确保各个进程彼此之间的相对独立。更确切地说，一个**独立型进程**（independent process）是不会影响另外一个进程的执行的，同时，一个独立型进程也不会受到另外一个进程的执行的执行的影响。而另一方面，我们有时会需要两个或多个进程相互协作。同样，准确地来讲，一个**协作式进程**（cooperating process）是一个会影响到另外一个进程的执行的进程，或者是一个会受到另外一个进程的执行的执行的进程。当我们试图开发系统时，我们有可能需要允许对应系统包含多个相互协作的进程，而且，有时我们会需要其中一部分进程运行在不同的机器上，有时则需要相关进程运行在同一台机器上。无论哪一种情况，这些协作进程将需要执行若干操作事项以便彼此可以成功协作来完成相应的任务。毫无疑问，它们需要**相互通信**（communication）。例如，一个进程可能在接收销售订单，然后它可能会把这些订单传递给另一个进程，而后者将开始办理运送商品的相关事务。同时，有关进程也可能需要同步它们的活动，以便它们不会因为在同一时间试图更新同一条信息而受到彼此的干扰。此外，当进程运行时，它们可能需要请求操作系统准许它们在一段时间内排他性地访问某类资源。事实证明，这种情况下可能会导致一种特殊类型的问题，即**死锁**（deadlock），操作系统需要对此类问题投入特别的关注并给以适当的处理。

本章将主要针对进程来展开讨论，但是，其中的大多数内容也同样适用于多线程。特别地，线程对内存的共享往往也会引发本章中所阐述的关于同步和死锁的诸多问题。然而，在一般情况下，线程不会使用在进程之间经常会用到的消息传递机制。

尽管我们未必期望把应用程序划分开来，但为什么我们有时候还是会那样做呢？在第9.1节中，我们将围绕这一问题背后的诱发因素展开讨论。接下来，第9.2节，我们阐述了协作式进程在相互通信时所使用的各种机制。而在第9.3节中，我们探讨了有关进程同步其相关活动的必要性，并讨论了这样做的一些机制。另外，当进程企图排他性地访问资源时，就有可能出现一种潜在的问题，即死锁。为此，我们在第9.4节中着重讨论了死锁。最后，我们在第9.5节对整章内容进行了归纳总结。

9.1 为什么会有协作式进程

在我们详细了解进程如何能够通信的相关细节之前，很自然地就会提这样的问题：为什么我们想把我们的系统分解成多个进程呢？而关于为什么我们想要开发应用程序是以若干部分运行的系统，则存在如下多方面的原因：

性能 (performance)。当我们设计一个系统时，我们可能有好多工作需要去做，即比一个较为廉价的处理器能够独立完成的工作量要多。从经济角度来说，花钱买上若干便宜的处理器，并让每个处理器上来分别运行相应进程的某一部分，要比买上一台可以完成整项任务的比较大的系统更为合算。而如果对应系统需要为大量用户服务，则采用唯一的一个系统可能无法为全部用户提供服务。

规模扩展 (scaling, 或称为伸缩性)。当我们最初开发一个应用程序时，我们不一定知道系统负载将会有多大。某项我们认为是很小的服务功能，可能在一夜之间就会引起轰动，因为它可能会要求为比我们原先设想要多出许多的用户来提供服务。谷歌 (Google) 搜索引擎就是一个很好的例子。无论该项服务的创始人有着什么样的梦想或愿望，非常令人怀疑，他们曾经想象过到 2004 年他们将需要 65 000 台服务器来运行有关的应用程序。显而易见，这项原因与性能问题密切相关，但又明显不同，它们并不是一回事儿。

组件购买 (purchased component)。我们可能想让我们的系统包含某些如万维网服务器之类的功能。想让我们发现白手起家去开发我们自己的网站服务器更为经济，那是不太可能的。我们最有可能是，使用一个现成的网站服务器程序，并设法对其进行修正——或许会通过编写所购买的服务器系统中那些用来动态创建系统所显示页面的相关部分——从而使其成为我们的系统。

第三方服务 (third-party service)。我们的系统可能会利用由另一个机构所提供的某项服务。一个典型的例子是登记信用卡费用。再次强调，想让我们开发出一个实现此项功能的系统能够与购买相应服务一样廉价，那是不太可能的。除非我们拥有非常多的交易要进行处理，我们才会考虑自己去专门开发此类系统，否则我们更倾向于付费给第三方来购买该项功能部件。

多系统中的组件。我们可能在构建一些执行类似任务的不同系统。我们往往不会为许多系统分别构建类似的模块并不得不去单独对它们进行维护，而是可以构建公共组件作为单独的进程来加以运行，并让各种不同的系统向对应公共组件提供和注入相应的事务。例如，现在有一家直接面向公众销售的公司可能要构建一个网站来支持客户从网上在线下订单以购买货物。该公司还可能在商店中有零售柜台，同时针对电视购物节目设立有电话销售组通过电话来接受订单，另外还有邮购产品组，他们都需要登记订单。这些系统可能各自以不同的方式接受订单，但均使用到了其他进程所提供的公共服务，首先是订单的验证，随后是采用客户信用卡方式的订单费用的支付，接下来是从仓库办理订单装运，同时还有库存状况的监测以及当发现商品似乎在低位运行时向对应的供货商下订单和续订商品。所有这些公共的组件都可以作为单独的进程来运行。

可靠性 (reliability)。当所有系统构建成一个整体并且都装到同一台计算机上时，那么该计算机的一个重大故障将会导致整个系统的终止。而如果这些系统是以模块化方式构建起来的，那么每个模块都可以有多个实例。还是回到前面的谷歌设计案例，如果 10 000 个系统中有一个系统出现了故障，那么搜索引擎系统将会继续运行。具体来说，可能有几个用户，他们的搜索已经分配到了那台故障服务器上。他们可能不得不去点击“刷新”按钮以成功获取搜索结果，但将可能完全不会意识到主机站点的某台服务器已经出现了故障。在谷歌的这个案例中，他们甚至把有关服务器分离来放置在不同的地方。这样即便是像火灾或洪水之类的物理灾难也不会把整个系统搞瘫痪。

信息的物理位置 (physical location of information)。即使是一家小公司，有时最终也会

拥有多处设施。通常这种情况出现是因为一家公司并购了其他的公司。例如，不管出于什么原因，我们可能最终会有多个仓库，而对于大部分的交易来说，我们通常想要在相应的地方分别构建一个库存系统。但是，出于其他的考虑，我们一般希望把该系统的某些部分放置在某个中心的地点。而鉴于批量折扣等理由，我们常常希望拥有一个统一的集中采购功能。如果我们已经设计了各种仓库库存系统以便让它们把库存请求提供给此集中采购系统，那么我们就可以从整体上把这看作是一个系统，而该系统的某些部分则分布在不同的物理位置。

志愿式启用型应用程序 (enabled application)。有极少数的应用程序拥有非常海量的计算要求，而这种计算要求的的确确在现有的计算机系统上是无法完成的。有时，这在一定程度上也是一个经济问题——是的，可以构建一部计算能力足够强大的机器来完成有关的任务，但想要解决这种问题的组织却往往承担不起相应的费用。有时我们只好等上几年。笼统地讲，可用处理器的计算能力每 18 个月就会翻上一番。而如果这条定律持续地发挥效用，那么我们可能很快就会拥有一台计算能力满足相应要求的机器。此类拥有海量计算要求的系统的一个例子如外星文明探寻 (Search for Extra-Terrestrial Intelligence, SETI) 项目所利用的系统。该系统接收由大型射电望远镜所记录的海量的数据，从中搜寻可能标明智能起源的数据模式。有关的数据规模相当巨大，因此，为了使这些数据可以利用当前可用的机器进行处理，就只好把它们划分成较小的数据集，并且把有关数据集分发给各种感兴趣的志愿者用户，而这些用户自愿让自己的计算机系统上的空闲时间被用来通过某种屏幕保护程序的相应机制进行对应数据的处理。不管你对这项努力的科学价值的看法如何，它却当之无愧地是采用这种技术的第一批系统中的一员。如果把它看作是一个松散耦合的系统，那么它就代表着目前世界上最大的一个计算机系统。不采用这种“分而治之”的方法，这些数据根本无法处理的。

183

9.2 进程间通信

由于上述的一种或多种原因，现在人们构建包含多个协作式进程的系统已经有一段时间了，并且此类系统的数量，无论从绝对数量还是在新的应用程序中所占的百分比，仍然在迅猛地增长。显而易见，如果我们要拥有一个由多个进程所组成的系统，那么这些进程之间必须得互相通信以共同完成相应的工作。然而，我们已经花费了大量的时间和精力，用来确保在同一系统上运行的两个进程不会相互干扰。因此，我们需要开发相应机制以允许进程间进行通信。当有关开发人员开始意识到这种需要时，他们中的那些在不同环境中的开发人员采用了不同的术语来看待这一问题，同时还运用了不同的工具来实现对应机制。对于使用系统网络体系结构 (System Network Architecture, SNA) 和同步数据链路控制 (Synchronous Data Link Control, SDLC) 协议的国际商业机器公司 (IBM) 的大型机客户来说，他们看待相关问题与使用诺勒有限公司网络操作系统的个人计算机用户并不相同，也与使用施乐网络系统 (XEROX'S Network System, XNS) 协议或数字设备公司分层网络体系结构协议 (Digital Equipment Corporation network, DECnet) 或传输控制协议 / 网际协议 (TCP/IP) 的 UNIX 或 VAX 用户有所不同。为此，如今出现了用于进程通信的几十种不同的机制。总体上来讲，主要存在两种截然不同类型的进程间通信 (InterProcess Communication, IPC) 机制。一方面，有许多消息传递机制，它们按照“消息”这个术语所指定的方式进行操作，也就是说，一个进程利用操作系统的某项功能把消息发送给另一个进程。而且，消息传递通常是在进程之间进行。另一方面，则是使用共享内存，其间，两个或更多个任务共享访问一块

内存。共享内存空间在同一个进程中的线程之间是隐含和不用言明的，但也可以用在进程之间。在具体讨论这两类通信机制之前，我们首先对所有相关机制的共同特征进行抽象，以便你在面对一种不同的通信机制时，你将会拥有一套有组织结构来识别其对应的重要特征。然后，我们再来介绍一些较为常见的通信机制。

9.2.1 通信机制的特性

可供进程用来进行通信的服务，一般可以按照如下不同特性加以描述：

- 一次可以使用单个信道的进程数
- 单向连接或双向连接
- 缓冲策略（没有缓冲区、单个缓冲区、 N 个缓冲区、无限个缓冲区）
- 面向连接的通信或无连接的通信
- 命名策略（名称、单向命名或双向命名、邮箱、端口）
- 多点传送、广播式传送、单点传送
- 可能有多个连接或单一连接
- 面向流的通信或面向消息的通信
- 异构通信或同构通信
- 同步通信或异步通信
- 持久通信或瞬时通信

所支持的进程数。在大多数情况下，在特定的进程间通信过程中，往往只会涉及两个进程。但在某些情况下，也可能会涉及许多进程，且这些进程之间同时进行了通信。例如，许多进程可能会同时共享了与（把记录写入日志文件的）某个进程之间的连接。

单向连接或双向连接。对于协作式进程来说，尽管仅仅开展单向的通信可能有点稀奇古怪，但单向通信信道则是稀疏平常的事情。通常情况下，单向信道发生的场合是，在两个进程之间可能建立了两个信道，但是这两个信道的通信方向是相反的。这种类型的通信机制往往适用于在一个方向上的通信任务要比在另一个方向上的通信任务繁重很多的情况。例如，一个进程在向另一个进程发送交易信息，而另一个进程只需向前一个进程发送回确认消息即可。于是，我们可能在第一个信道上需要许多大容量的缓冲区，但在第二个信道上则仅需要少得多或小得多的缓冲区，即只需要较低的带宽。

缓冲策略。根据可用的缓冲区数量，存在没有缓冲区、单个缓冲区、 N 个缓冲区以及无限个缓冲区 4 种不同的通信信道缓冲区处理情况。第一种情况是指，没有缓冲区是可以被通信的两个进程都能够访问的。两个进程必须同时访问信道，以便一个进程可以把消息直接发送给另一个进程。第二种情况是指，只有一个缓冲区是可以被通信的两个进程都能够共享访问的。在这种情况下，首先由发送进程把一条消息放入缓冲区中，然后告知操作系统该缓冲区已准备好了消息。接下来，操作系统将告知接收进程有新的消息放在了缓冲区。于是，接收进程将消息从缓冲区中取出，并告知操作系统，发送进程可以发送下一条消息了。不言而喻，单个缓冲区的情况看起来可能只是 N 个缓冲区情况的一种特例。然而，在单个缓冲区的情况下，我们可以使用非常简单的机制来同步有关进程，相关进程始终知道要使用的是哪个缓冲区，并且只需要做好缓冲区现在是否可供发送进程放入新消息的协调工作（译者注：其实，同时还需要做好缓冲区现在是否可供接收进程从中提取消息的协调工作。但是考虑到，缓冲区可供发送进程放入新消息和可供接收进程从中提取消息的这两种状态恰好相反，

故而可以优化整合成一种协调工作。也就是说,当缓冲区现在可供发送进程放入新消息时,肯定不能供接收进程从中提取消息;相反,当缓冲区现在不可供发送进程放入新消息时,肯定可以供接收进程从中提取消息)。在 N 个缓冲区的情况下,我们需要协调更多的信息。每个进程的通信信道机制必须知道是对应于哪个通道的缓冲区、哪些缓冲区包含了消息以及哪些缓冲区没有包含消息。我们将会在第9.3.9节中进一步讨论相关问题。最后一种情况是指,相应信道机制拥有一些外部存储器可以用来扩展缓冲空间,且从本质上讲可以把缓冲区数量扩展到无限多个。举例来说,在假脱机系统中,使用磁盘文件来保存有关消息流,直到有打印机可供使用和将其打印出来为止。从实际使用的角度来讲,发送进程可把有关缓冲区看作是没有穷尽的。

面向连接的通信或无连接的通信。一个通信信道可以是面向连接的,也可以是无连接的。有时,通信的进程之间可能需要建立一个复杂的对话。在这种情况下,它们很可能要建立一个它们将在交互期间使用的连接。这就好比是通电话,一个人在打电话给另一个人。相关术语一度是源自于采用实际的物理连接在设备之间所建立的连接。但是现在,有关连接很可能是逻辑连接或虚拟连接。有时,一个进程只是要把消息发送出去,而并不关心其他进程在监听接收什么。这样的应用程序例子具体可能是,某家公司的一台服务器正在广播股票购买信息,以便客户端可以在需要时能够接收到有关的信息。这也可以比作是无线电广播方式,可能会有数百万的听众,也可能一个听众都没有。

[185]

命名策略。当发送进程和接收进程彼此之间要进行通信时,它们往往需要某种机制来识别对方。这称为命名策略。在最严格的情况下,通信的双方进程都必须有明确的名字来辨别对方。大多数情况下,有关名字就是用来运行对应程序的可执行文件的名称,但是这些名字一般能够以其他方式和正在运行的进程关联起来——在某些系统中,作为进程标识符的数字也可能被用作辨别的名字。这种特定的命名机制具有误差最小,即最不容易引起错误的优点,但是需要尽最大努力去加以维护。因此,这种命名机制通常只有在通信的双方进程都是由相同的开发人员负责设计开发并且只有一个发送进程和一个接收进程的情况下才会适用。在稍加宽松的模型中,消息发送者必须指定接收进程的名称,但接收者则可以接受来自任何发送进程所传输的消息。第三种模型是发送进程和接收进程就它们即将共同使用的某种其他引用方法达成一致,具体例子包括邮箱编号和TCP/IP端口。

进程间通信的另外一种特性是指,有关消息是单点传送(unicast),还是多点传送(multicast,或称为多播或组播),亦或广播式传送(broadcast)。单点传送的消息仅仅被发送给接收者进程。因此,如果是许多进程在进行协作,那么,为了让所有进程都接收到某条消息,可能需要发送许多条消息,即许多次发送相同内容的消息。尽管如此,单点传送的消息还是私有性质的。广播式传送的消息发送出去之后,(在给定环境中的)每个进程都可以接收到。这样的例子,比如周期性地发送时钟更新消息的时间服务器,任何进程都可以读取到相关消息。令人遗憾的是,广播式传送的消息必须被所有进程接收和处理,无论对应进程对有关消息是否感兴趣,故而这种方式可能造成资源的极大浪费。不过话又说回来,某些消息,譬如系统关机请求,确实可能需要被所有的进程所接收。多点传送的消息旨在针对一组接收者进程进行发送。有时,这是出于安全原因的考量,所以相应组内的成员资格可能会受到限制。但是,还有些时候,多点传送的组别创建只是为了能够给那些对有关消息不感兴趣的进程节省一些资源——证券报价应用程序可能就是一个很好的例子。

可能有多个连接(或称之为多重连接)或单一连接。大多数情况下,允许两个进程之间

只有单一连接就已经足够了。但是，有些时候，期望用于传送数据消息的信道和用于传送控制消息的信道是分离和独立的。文件传输协议（FTP）就是以这种方式使用两个连接的标准的例子。另外还有多个并行数据连接的情况。在使用超文本传输协议（HTTP）1版的一些网络浏览器/服务器连接中所使用的机制就是多个连接的一个例子。在该协议中，单个信道只能从服务器检索一个对象，之后服务器将关闭对应连接。为了加快进程的运行效率，浏览器往往会检索主页、对其进行解析并试图从中查找需要显示相应页面的其他元素，从而接下来会打开尽可能多的连接，准确地讲是，每当有对象要检索时就再打开一个连接。（实际上，客户端往往一次只会打开有限数量的连接，从而防止服务器被拖垮和陷入瘫痪。）

186

面向流的通信或面向消息的通信。对于一些应用程序而言，通信链路支持离散消息（discrete message）的想法是十分重要的。负责发送的应用程序发送一个数据块，并且该数据块将以相同的方式（即作为离散消息）呈现给负责接收的应用程序。这些数据块可以是固定长度的，也可以是可变长度的，具体取决于相应的实现。其他的应用程序则不识别数据中的块，而是把对应通信看作是从发送方流向接收方的数据流（stream）。一个简单的例子如远程登录客户端应用程序。用户在键盘上点击的每个按键将会从客户端直接发送给服务器，而不考虑对应的内容是什么。某些按键（例如，CTL/C）的处理可能会优先于其他的按键，并且有关协议可以将许多按键捆绑到一起进行传输，从而使传输开销最小化。不过，一般而言，这些按键是作为连续的信息流进行发送的。

异构通信或同构通信。一些通信系统往往假设消息的发送者和接收者是运行在相同类型的硬件和相同的操作系统平台上的。最鲜明的情况是，假设两个通信进程运行在同一台机器上。而其他的通信系统则没有做出这样的假设。为此，后一类通信系统可能需要尝试处理与信息表示相关的一系列问题。因为，不同的系统是以不同的格式来存储信息的。通常情况下，这是出于硬件方面的考虑，比如说整数的存储。在英特尔 80 X 86（Intel 80 X 86）系列硬件平台上，数的最高有效字节（Most Significant Byte, MSB）被存放在较高地址的存储器单元中。但在大多数其他的硬件平台上，数的最高有效字节则被存放在较低地址的存储器单元中。（这被称为“小端字节序/大端字节序”问题，即“little endian/big endian”问题，相关提法源自于《格列佛游记》的有关故事。）如果一个系统在可能以不同格式实现整数的平台之间发送消息，那么该系统或许想以一种通用的方式来解决有关问题。例如，一个通过远程过程调用（Remote Procedure Call, RPC）方式进行调用的子例程可能需要针对相关参数执行算术运算，因此发送方和接收方将需要以对应用程序透明的方式来解决这一问题。另一方面，文件传输协议（FTP）例程只是用来对文件进行传输的。相关内容的重新编排并不是通信机制所关注的方面。尽管如此，文件传输协议例程可能需要考虑文件命名约定的差异。也就是说，在发送方系统上的一个文件的名称可能并非是接收方系统上的一个合法的文件名。还有一些关于格式编排问题的例子没有涉及硬件，而是与相应的编程实现语言有关。例如，字符串可能在 C 语言中以这种方式进行存放，但在 BASIC 语言中则以另一种方式进行存放，因此，拥有不同语言编写的组件的系统可能必须得在这些格式之间进行数据的转换。（当然，这种问题无论是对于运行在单个处理器上的单个程序，还是对于多个进程，都是可能的。）还有可能遇到的另一个问题是，一条消息的某个参数有可能是一个内存地址——如可能是指向某个错误例程的指针。显然，如果这样的参数被直接传递到了在另一个平台上运行的进程，该参数将是毫无意义的。如果我们想要把内存地址作为参数进行传递，我们将必须创建某种其他的通信机制来予以相应的支持。

同步通信或异步通信。对于大多数高级语言而言，当程序从一个文件中读取记录时，有关功能所对应的模型通常是，在下一条指令被执行时，相应的读操作已经完成。这种输入/输出模型称为同步输入/输出（synchronous I/O）或阻塞式输入/输出（blocking I/O）。
[187] 如果一个进程在执行读操作的同时还要参与完成其他的任务，那么该进程可以选择以异步（asynchronous）读取方式或非阻塞式（nonblocking）读取方式来发出读操作请求。在这种情况下，读操作指令将立即返回到对应程序，并且读操作将会独立地执行。最终，程序将会探询有关读操作是否已经完成，具体方法取决于相应的程序设计语言和操作系统。通信信道也是类似的。一个进程可能想要检查某信道找寻消息，但是如果没有可用消息，该进程会继续执行。因此，对于某些通信机制而言，一个接收进程可以发出一个针对相应信道的异步读操作请求，并且如果没有信息要读取，那么操作系统将带着一个表征没有数据传送的结果从该调用返回且该进程将不会等待。类似地，如果没有缓冲空间可以用来接收有关的消息，一个异步写操作请求也可能被拒绝和抛弃。

持久通信或瞬时通信。在最简单的情况下，发送进程和接收进程必须同时运行以便于它们交换消息。如果其中一个进程不可利用，那么相应通信系统便无法正常运作。这种类型的机制称为瞬时通信。在其他的系统中，操作系统将会保留发给一个当前并不运行的进程的那些消息，或者会传送由一个当前并不运行的进程所发出的那些消息。这样的通信服务称为持久通信。

9.2.2 进程间通信系统的例子

在最简单的情况下，一个发送进程可能只需要传递一个最少数量的信息，即一个二进制位，其一般用来表征某事件已经发生。对于这种情况，相关进程可以利用下一节中将要描述的同步机制。然而，在大多数情况下，进程需要发送比单个二进制位更多的信息，因此它们将使用更为精巧的机制，即进程间通信（InterProcess Communication, IPC）系统来发送整个消息。

一种被广泛使用的进程之间消息交换的方法是管道（pipe）的运用。管道本质上就是一个循环缓冲（或循环队列），并且一个进程把数据放入其中，而另一个进程从中取出数据。这两个进程可以利用系统调用来放置和提取数据。这种机制使有关进程无须再去担忧同步问题。（我们很快将会讨论这一问题的本质。）操作系统将会寻找一个放有消息的所谓满的缓冲区或者一个没有有效消息的所谓空的缓冲区，不过主调例程需要意识到相应调用可能没有成功。例如，如果对应缓冲区已满，那么尝试将数据放入该缓冲区的发送例程将会被阻塞。通常情况下，一个接收例程可以调用非阻塞式系统例程来尝试从缓冲区中读取数据。如果有数据可用，则将对数据返回。而如果缓冲区中没有数据，那么对应调用也将立即返回，但会有一个返回码用来指示没有消息被读取到。往往还会有阻塞式类型的读操作。如果一个接收者除了等待传入的信息之外而无其他事项可做，该接收者就可以使用阻塞式读操作。管道最初出现在 UNIX 系统中，并且在 UNIX 和 Linux 系统中，管道是字节流而不是离散消息，同时相应管道是单向的信道。在 Windows 系统实现中，管道可以是字节流，但是它们也可以用于发送消息，并且相应管道是双向的。

关于管道的另一个问题是如何在一开始对它们进行设置的问题。在某些情况下，发送进程和接收进程必须都有明确的名字来辨别对方。这在通常情况下是不受欢迎的，因为与其他方法相比，这将更难以实现和维护。而有些时候，接收者可能并不在乎谁正在发送，但发送

者必须明确指定接收者。一个把消息写入日志文件的应用程序可能会同时向多个客户端提供此项服务。该应用程序将会接收到关于是哪个客户端发送消息的一个标示信息，并且还可以把相关信息记录到日志中。还有一些时候，发送方提供的不是接收进程的名字，而是关于接收进程的某种其他的引用标记，这有时被称为**邮箱**（mailbox）。在这种情况下，发送者正在寻找一种可以由许多不同进程所提供的服务，并且发送者并不介意哪一个进程在提供相应服务。**命名管道**（named pipe）是此类机制的一个例子。

现在，**套接字**（socket）机制已经使用有很长一段时间了，因此，这种机制的优势之一就是拥有许多兼容性的实现——套接字机制几乎在每种操作系统上都有实现和都可以使用。套接字是专门被设计在标准网络层上运行的。在大多数情况下，这一层的下面是网际协议（Internet Protocol, IP），而上面为传输控制协议（Transmission Control Protocol, TCP）或用户数据报协议（User Datagram Protocol, UDP），不过也存在其他的实现。客户端（发送方）主机需指定服务器（接收方）主机的名字，同时还应指定在接收方主机上的特定的套接字（有时称为端口）。这些只是逻辑意义上的指定，而不是对硬件端口的引用。在许多情况下，这个套接字将会是一个众所周知的数。大家都很熟悉的套接字（在1024以下）常常已经被有关标准化组织分配给了相关的标准化协议。而较大的套接字编号则被预留用于未被标准化的应用程序。当客户端尝试连接到相应的服务器时，客户端将会被分配一个套接字编号以供其所使用。与用于管道中相对简单的单向缓冲机制不同，套接字支持更为复杂精细的模型。一旦客户端和服务端之间建立了连接，那么它们所使用的协议将会由对应的应用程序来决定。无论是客户端还是服务器都可以在任何时间发送消息。对于某些应用程序来说，应用层协议是已被标准化的，但是，新的应用程序可以设计其所需要的任何类型的协议。应用程序还可以将现有的协议用于各种不同的目的。例如，应用程序通常都支持超文本传输协议，因为许多防火墙都被设置为放行这种协议的信息流。通常情况下，服务器则通过执行一系列操作系统调用来创建套接字，然后开始等待与相应套接字的传入连接。而如果服务器提供了诸如文件传输协议之类的复杂服务，那么服务器往往会启动一个单独的线程来负责处理与每个客户端之间的交互。但如果相关的服务非常简单，譬如说只是提供当日报价的服务，那么服务器可以只是发送一条消息，然后就可以中断对应连接。

套接字设计的另一个优点是，服务器端和客户端可以位于同一台机器上。这在开发新的应用程序时特别方便。同时，这也意味着同样的程序，无须对任何程序（无论是客户端程序还是服务器端程序）进行任何的修改，就能既可以为本地的客户端提供服务，还可以为远程的客户端提供服务。这是一个非常整洁干净的模型，没有其他机制的某些复杂性。当然，这也意味着，对于本地的客户端来说，正在完成的大量的工作并不是必需的。如果要考虑性能问题，并且对应系统始终是在同一台机器上运行客户端和服务端，那么应当使用更为高效的机制。

一种持久的通信机制称为**消息队列**（message queuing）。此类系统往往会创建被指定名字的消息队列。具体而言，一个希望把消息写入队列的进程将执行有关的操作系统调用，同时传入消息和用于放入消息的队列的名称作为相应的参数。而一个希望从队列读取消息的进程同样也需要执行有关的操作系统调用，同时传入一个空的缓冲区和对应队列的名称作为参数。这两类进程可以同时运行，也可以不是同时运行的。其他的进程或系统实用程序通常被用来创建和销毁相应的消息队列。这些队列经常被保存在辅助存储器中，从而确保其持久性，因此这种通信机制的管理开销也是很大的。

9.2.3 共享内存系统的例子

对于许多操作系统而言,在同一台机器上运行的两个(或多个)进程有可能会要求操作系统允许它们共享访问某一内存块,这种技术被称为共享内存(shared memory)。相关内存共享实施过程通常是这样的,一个进程调用操作系统以请求分配(某指定长度的)一段内存,并获得相应内存分段的一个名字。而其他希望共享此段内存的进程必须要知道该内存分段的名字,且有关进程通过向操作系统提供这一内存分段名字来请求访问对应内存分段。所有相关进程的内存地址设置应当予以相应的调整,以便它们可以访问对应的共享内存块。这一机制的确切内容将在第11章中具体讨论。此外,一些应用程序非常简单,并不需要复杂的同步操作来确保相关两个进程不会相互干扰。但是,也有一些其他的系统则可能需要更为复杂和精巧的同步机制来控制对指定共享内存块中的数据的访问,这一主题及相关内容将在下一节中详细阐述。需要注意的是,虽然独立的进程之间必须使用此类复杂精巧的机制来实现内存的共享,但是根据定义,同一进程中的线程之间则始终可以共享它们的内存。

内存映射文件(memory mapped file)是共享内存的一种特例,这是对共享内存技术的一种轻微的改动。具体来说,首先,发起者进程调用操作系统,同时传入辅助存储器上的一个文件的名字作为参数。然后,操作系统查找定位对应文件,并在主调进程中分配空间使其包含整个文件。尽管如此,该文件并不会立即加载到内存中。当共享文件的某些部分第一次被访问的时候,硬件将会发信号通知操作系统,从而把文件的相应部分加载到内存中,并恢复对应进程的运行。这种机制将在第11章中展开更为全面详尽的描述。

9.3 同步

9.3.1 相关问题

关于为什么进程需要通信以及它们可以用来通信的一些机制,我们现在已经有了一定的认识。接下来,我们将把注意力转移到当两个进程想要共享内存中的一块数据时可能发生的问题上来。考虑下面的例子,有两个进程A和B,它们正在使用一个缓冲区进行通信并尝试更新一个共享记录计数器 X (初始值为8)。进程A把一条记录放入缓冲区中,并试图通过对 X 的加1操作来使共享记录计数器递增。进程B则从缓冲区中提取一条记录,故而它应试图通过对 X 的减1操作来使共享记录计数器递减。为此,进程A拥有一条指令 $X=X+1$,同时进程B拥有一条指令 $X=X-1$ 。当进程A、B顺次执行记录放入操作、记录提取操作以及这两条指令执行后,我们期望 X 的最终取值仍然为8。然而,这里存在一个潜在的小问题。

通常情况下,上面我们所说明的两条高级语言命令可分别分解成三条独立的机器指令,即寄存器加载指令、加法或减法指令以及存储指令。考虑图9-1所示的执行过程。首先,进程A将变量 X 的取值加载到寄存器A中,于是,寄存器A中就包含了数值8。这时候,假定进程A因为自己的时间片已经用完所以被中断。相关寄存器内容被保存在进程A的进程控制块中。此时,进程B获得了一个时间片,故而将变量 X 的取值加载到寄存器A中。为此,寄存器A的内容同刚才进程A的操作结果相比并没有发生任何改变。接下来,进程B从寄存器A的内容中减去1,从而使之变成了7,然后将其存放在变量 X 对应的内存单元中,这样就使变量 X 取值成为7。进程B继续去执行其他的操作。最终,进程A会被再次调度和获得另一个时间片。进程A的寄存器的内容根据其进程控制块进行了恢复,故而现在寄存器A的内容再次变成了8。进程A对寄存器A执行加1操作,从而得到9,然后将

其存放在变量 X 对应的内存单元中，这样就使变量 X 取值变成了 9。显然，这根本不是我们所期望的最终结果。更为糟糕的是，这问题是与时序相关的。也就是说，存在一个非常小的时间窗口，期间才可能发生此类问题，而几乎所有的时间，这两个进程都会非常融洽和恰到好处地共享这个变量，假设这是它们对该变量所做的唯一的修改。这种问题因为其间歇性特征，所以很难调试，称之为**竞态条件**（race condition）。进一步说，竞态条件，或称为**竞争险象**（race hazard），是指设计方面的一个缺陷，相关系统输出取决于其他事件的次序或时序。对于多处理器系统而言，当进程 A 运行在一个处理器上而进程 B 运行在另一个处理器上时，此类问题也可能发生。无论是什么原因导致了这类问题，我们都需要一种解决方案。虽然多处理器系统在高端服务器之外并不常见，但是在单个芯片内拥有多个处理器已经成为当前这一代处理器芯片的焦点。因此，这类问题将变得更为普遍。

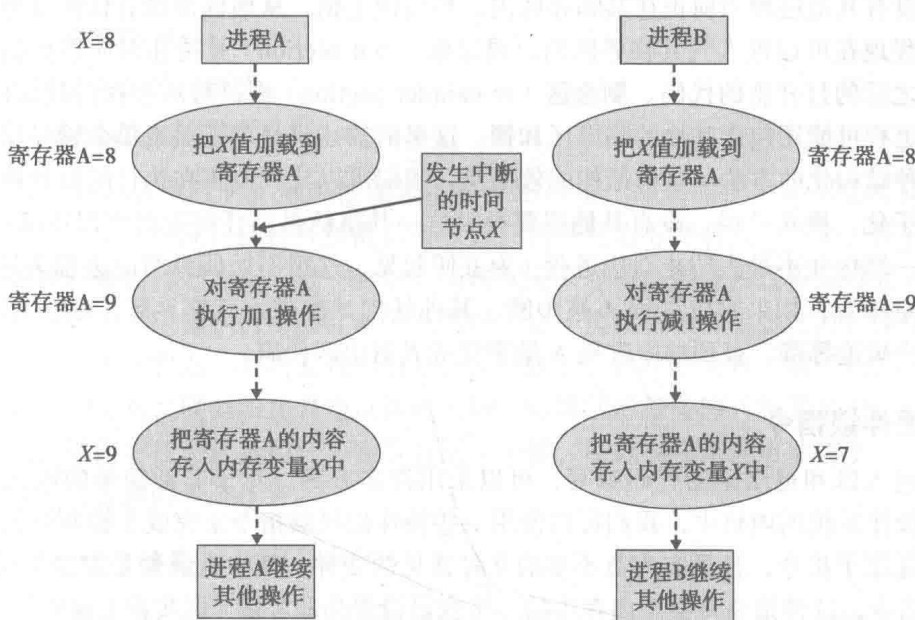


图 9-1 两个进程对一个共享变量的更新

9.3.2 原子操作

我们所需要的技巧是让那些操作**原子化**（atomic）。原子化意味着我们所做的操作是不可分割的——特别地，该操作不能被想要更新同一信息的另一个进程所打断。一种可能的解决方案是把对应指令编译成一条不可被中断的指令。例如，一些机器可以对内存中的某个变量执行加 1（或减 1）操作而不用把对应变量加载到寄存器中。但是，当我们用高级语言编写我们的程序时，我们常常不想为这样的硬件细节所挂虑。我们往往希望能够把我们的程序移到一台不同的可能没有此类指令的机器上。因此，我们必须得使用一种更为通用的解决方案。

191

9.3.3 锁与临界区

有时，我们的进程只是在共享一个变量，而有时，它们则共享着一个较为复杂精巧的结构。有时，我们只是在执行单一的操作，而有时，我们则会执行较为复杂的多个操作。有时，我们仅仅有两个进程试图去共享一个资源，而有时，我们则会有很多进程在共享一个资

源。有时，我们在尝试共享一个拥有多个实例的资源，而有时，我们则在尝试共享一个拥有仅仅一个实例的资源。我们将要使用的这种通用的技术就是，用一种称为锁（lock）的特殊类型的变量来控制对共享变量的访问。锁有时也被称为**互斥机制**（mutex），因为它可以提供对锁保护下的数据项的互斥访问。当我们审视一个使用锁的进程时，我们可以认为相应的程序拥有 4 个部分，如图 9-2 所示。

```

Main() {
    <进入区> /* 确保对应锁是打开的 */
    <临界区> /* 操纵处理被共享的数据 */
    <退出区> /* 指示对应锁是打开的 */
    <剩余区> /* 任何其他操作 */
}

```

图 9-2 一个操控共享内存的进程的组成

具体而言，**临界区**（critical section）是指对应进程中用于操纵处理那些可能也被其他进程所操纵处理的信息的代码部分。**进入区**（entry section）是指对共享信息上锁的代码——其首先确认没有其他进程当前正在其临界区内，然后锁上锁，从而保证没有任何其他共享对应信息的进程现在可以进入到其临界区内。**退出区**（exit section）是指在对应进程执行结束临界区代码之后的打开锁的代码。**剩余区**（remainder section）则是对应进程的其余代码部分。注意这个进程可能还包含其他的临界区和锁。这里的描述只是审视关于单个锁操作的各组成部分的一种结构化的方法。这种结构的效果是，我们可以让我们正在执行的针对被共享信息的操作原子化。换句话说，没有其他想要操纵这一共享信息的任何进程可以中断该临界区。但是，这一结构并不是去防止对应进程（为方便起见，这里不妨假设对应进程为进程 A）被中断。这意味着，如果对应进程 A 被中断，其他任何试图进入其临界区（对应于这个变量）的进程将会被迫等待，直到对应进程 A 结束完成其退出区代码。

9.3.4 硬件锁指令

关于进入区和退出区的代码编写，可以采用许多方案。对于非常简单的嵌入式操作系统或者在操作系统的内核中，我们可以使用一些特殊的机器指令来完成上锁和开锁。这些指令本身就是原子指令，且只有为数不多的几种常见的变种。其中有一种是**测试 - 设置**（Test and Set）指令，这种指令将会把内存中的一个变量设置为非零值（即实施上锁操作），同时返回一个结果以告诉我们在我们执行这条指令之前是否已经上锁。如果在我们执行这条指令之前已经上锁，那么我们就没有获得访问被锁定的临界区的权限，所以我们只好重新尝试这一操作指令：

```
while (TestAndSet(MyLock)) ;
```

注意，这里 while 循环语句的循环体是空的，所以这个循环的内部实际上并不会执行任何操作，直到当该指令执行返回未上锁的结果时才会终止循环。这种类型的编码有时被称为**自旋锁**（spin-lock）或**忙等待**（busy-waiting）。另一种常见的原子指令是**对换**（swap）指令，该指令可以在一步里完成两个变量取值的交换。从对换指令可以把任何值放入锁变量的意义上来看，这一指令的功能比测试 - 设置指令要稍微强大一些。除此之外，二者是等效的。还有另外一种类似的指令称为**取 - 加**（fetch-and-add）指令，该指令从内存中获取一个用作锁的整数，同时针对该整数执行加 1 操作并将其写回到同一内存位置。从 80486 处理器起，在英特尔处理器中已经开始使用以这种方式运作的 XADD 指令了。选择这些原子指令的哪一种进行实现，具体取决于硬件设计问题。如果采用前两种指令，我们的进程的退出区将只需把对应的锁变量设置为假（即零）即可。而通常情况下，把零值存入内存位置在任何硬件上都是一条原子指令。

9.3.5 信号量与等待

如果一个进程真的包含了上面我们所给出的使用测试 - 设置指令的 while 循环，那么在其等待的过程中，将会是在浪费处理器周期。因此，在大多数情况下，应用程序往往不会使用这些指令来实现有关的进入区代码。进一步说，对于进入区和退出区的代码设计来说，应用程序常常会采用操作系统调用请求的方式。通常情况下，在这些调用中所使用的变量被声明为信号量（semaphore）。与我们已经描述的简单的锁机制相比，信号量可能要更为复杂一些。锁机制永远是二值状态，但信号量通常要更为一般化。鉴于此，简单的信号量经常被称为二值型信号量（binary semaphore），并且只支持互斥访问锁定功能。

虽然不同的操作系统和语言对这些例程往往会使用许多不同的名称，但一般来讲，关于进入区的系统调用可以采用如下简单的形式：

```
wait (MySemaphore)
```

而退出区的系统调用则为：

```
signal (MySemaphore)
```

当我们调用 wait 例程时，如果被锁定的资源不可用，那么操作系统不是将我们的进程置于循环过程，而是使我们的任务脱离运行状态并将其置为等待状态。于是，该进程只有等待，一直到对应锁被打开为止。进一步说，当现在进入对应临界区的任务通过其退出区并针对我们正在等待的对应锁执行 signal 系统调用时，这一情况将会发生。此时，操作系统将让我们的任务离开等待状态，并将其置为就绪状态。随后，操作系统将会赋予我们的任务进入临界区和上锁的权利，同时防止其他试图进入同一临界区的任务闯入临界区中。正如我们前面所提到的，可能有任意数量的任务在等待同一个锁，所以操作系统可能需要一个队列用于存放等待每个信号量的所有任务。

193

9.3.6 计数型信号量

然而，更为常见的情况是，每个信号量经常会关联有一个正整数的计数器。这样的信号量有时被称为计数型信号量（counting semaphore）。它们通常用来控制针对一组同类资源（例如缓冲区中的记录）的访问。与计数型信号量相关联的计数器一般被初始化为相应可用资源的数量。在应用程序中，关于计数型信号量所使用的代码通常与关于二值型信号量所使用的代码是相同的。当一个进程想要访问某类资源的一个实例时，它会调用如前所述的 wait 例程。如果与对应信号量相关联的计数器取值为零，那么说明对应资源没有实例可以使用，于是相应的请求进程被阻塞。而如果对应的计数器取值大于零，那么说明对应资源有实例可以使用，于是对应计数器将会递减以表明一个实例正在使用中。当进程调用 signal 例程时，对应的计数器将会递增，同时相应的可用资源将会被分配给某个正在等待的进程。

计数型信号量还可以用于对文件访问的同步，其间，相关进程往往会执行大量的文件读操作以及少量的文件写操作。一般而言，可以允许许多只执行读操作的所谓读者进程同时运行，但是当有进程试图写操作对应文件时，我们可能并不希望读者进程处于活动状态。因此，我们可以采用计数型信号量来支持多个读者进程同时读操作文件，且只有当读者进程的计数值变为零时，才会允许一个包含写操作的所谓写者进程来写操作相应文件。一旦一个写者进程试图访问对应文件及相应的锁（译者注：用于实现写者进程与其他进程之间的互斥访问），我们将不允许任何其他读者获得文件的访问权，直到对应写者进程访问了被锁定的

文件并完成其工作为止。

9.3.7 同步与流水线体系结构

当多个处理器存在于单个系统中并且相关处理器具有流水线体系结构时，处理器执行指令的次序可能会发生混乱。这便可能导致诸如测试-设置指令之类的同步指令出现时序问题。因此，往往会提供某种机制以允许应用程序（或操作系统）发出一种命令，来限制处理器按照相关指令在程序中的次序来执行某一特定的指令序列，从而避免此类问题的发生。

9.3.8 对称多处理系统中的同步

我们曾经在前面的章节中说过，计算机系统发展的趋势是包含多个处理器，特别是多个处理器被合并和融入单个集成电路中的所谓多核处理器。此类系统要求操作系统能够管理多个处理器的资源。首选的解决方案被称为对称多处理架构（Symmetric MultiProcessing, SMP）。在这种体系结构中，操作系统被设计为可以运行在任何处理器上。（一种替代的体系结构被称为非对称多处理架构。其中，一个处理器上运行操作系统，而其他的处理器上则只运行应用程序。这种架构现在已经很少见到。）

在各个独立的处理器上运行的操作系统的多个执行流，可能会尝试同时引用相同的数据。为了避免这种情况，对称多处理操作系统将会使用锁。我们曾经说过，用户程序并没有使用自旋锁，因为它们会浪费掉宝贵的处理器周期且相应时间长度未知。然而，在操作系统内部，我们大致会知道我们只会在非常短暂的预定长度的时间内锁定。同时，我们无

[194] 法非常合理地去执行一个操作系统调用到 wait 例程中，因为这时我们已经是在内核中了！因此，尽管会浪费掉处理器周期，操作系统内核代码通常情况下还是会使用硬件的自旋锁机制。

然而，这就带来了一个有趣的硬件问题。设想一下若干处理器中的某个处理器希望将某个数值写入内存中的情况。大家都知道，为了保护临界区，会使用一个锁。同时，信号量是存放在共享内存中的，且每个处理器也可能在其高速缓存中拥有这一信号量的数值。或许你已经能够看到问题了一一现在，有很多的处理器在使用那些相同的原子指令，并且很可能是同时使用。这便要求在对称多处理架构中，共享内存的各个处理器对于共享操作要变得更加精明些。为此，一种常见的做法是，各个处理器均通过窥探（snooping）内存总线来监视针对共享内存的相关操作。对于处理器硬件来说，这是额外的工作，但却是非常值得付出的努力——采用多个处理器带来的提速潜力是相当巨大的。另外，高速缓存硬件也必须更加精明些，因为每个处理器都可能在其高速缓存中拥有对应信号量的一个副本，并且如果一个处理器改变了该信号量的取值，那么所有其他的副本也必须同时被更新。对于处理器架构设计人员来说，这些都是非常重要的问题，并且受到了非常广泛的争论。

9.3.9 优先级倒置

当较低优先级任务持有较高优先级运行的任务所需的共享资源时，就可能出现一种所谓**优先级倒置**（priority inversion）的情况。这种倒置将导致高优先级任务的阻塞，直到相应资源被释放为止。其实质是颠倒了这两个任务的优先级。如果其他某个没有使用对应共享资源的中等优先级的任务试图运行，则该中等优先级的任务将先于低优先级任务和高优先级任务而执行。一般而言，优先级倒置往往不会引起巨大的危害。其后果只不过是，高优先级任务

的延迟未被察觉，而最后低优先级任务还是会释放相应共享资源的。但是，优先级倒置也可能引起一些严重的问题。进一步说，如果高优先级任务被延迟足够长的时间，那么可能导致定时器的触发和操作系统的重启。火星探路者（Mars Pathfinder）项目就曾发生过这样的优先级倒置问题，并导致相应系统好多次执行重启操作。至少来说，优先级倒置可能使系统看起来会不合常理地慢下来。另外，低优先级任务之所以拥有较低的优先级，常常是因为，只要它们的工作最终可以完成，它们在哪个特定时间段（或称为时间帧）完成并不重要。然而，高优先级任务则经常有严格的时间限制，这类任务往往和用户界面一起运作或者是在软实时任务上运作。因而，优先级倒置可能会降低系统的响应速度。

9.3.10 经典问题

有一个问题在操作系统中经常会出现，称为生产者 - 消费者问题（producer-consumer problem）或有界缓冲区问题（bounded-buffer problem）。这是多进程同步问题的一个例子。该问题涉及至少两个进程，其中一个是数据的生产者，而另一个是数据的消费者，并且它们之间共享着一个公共的大小固定的缓冲区。生产者进程的任务是连续不断地生成数据块并将它们放进缓冲区中。与此同时，消费者进程则通过每次从缓冲区提取一块数据的方式来消费数据。需要指出的是，如果缓冲区中已经放满了数据，那么生产者就不应该再向缓冲区中添加数据。相应地，消费者也不应该试图从空的缓冲区中去拿走数据。这一问题的一种解决方案参见下面给出的程序伪代码。该解决方案适用于多个消费者和多个生产者，不过在这里，我们将就一个生产者和一个消费者的情形具体展开讨论。

195

```
semaphore mutex = 1
semaphore full = 0
semaphore empty = BUFFER_SIZE
```

```
procedure producer() {
  while (true) {
    item = produceItem()
    wait(empty)
    wait(mutex)
    putItemIntoBuffer(item)
    signal(mutex)
    signal(full)
  }
}
```

```
procedure consumer() {
  while (true) {
    wait(full)
    wait(mutex)
    item = removeItemFromBuffer()
    signal(mutex)
    signal(empty)
    consumeItem(item)
  }
}
```

生产者的解决方案是，如果缓冲区已经填满，就阻塞自己。每次消费者从缓冲区中拿走一块数据时，它都会发信号通知生产者可以再次往缓冲区中填放数据了。同样，消费者如果

发现缓冲区是空的，也会阻塞自己。每次生产者将数据放入缓冲区中时，它都会发信号通知消费者可以继续从缓冲区中提取数据了。计数型信号量 `full` 表示当前缓冲区中放有数据的数量，计数型信号量 `empty` 则表示缓冲区中还可填放数据的数量，而二值型信号量 `mutex` 则用于建立互斥访问机制。

9.3.11 管程

虽然表面上看起来似乎并不困难，但是，锁和信号量的使用则是程序设计中非常容易出错的部分。为了使关于上锁和开锁的相关处理更为健壮，一些高级语言中已经引入了用于表达同步需求的相应机制。这种机制被称为**管程**（`monitor`）。管程并不是操作系统的结构体，而是以一种相对不容易出错的方式对操作系统结构体进行打包处理的方法。进一步说，一个管程就是嵌入了互斥和线程同步功能的某样东西。这些功能由编程语言进行定义，以便编译器可以生成正确的代码来实现相应管程。虽然管程在不同的语言中采取了不同的形式，但是关于管程，还是有一些我们可以介绍的通用的内容。

一个管程往往与对应语言中的某种东西（例如过程或类）相关联。而互斥机制则常常与进程相关联。无论在任何时候，只能有相关联进程的一个线程可以在管程中执行。一个管程进程在执行其他任何操作之前，通常会尝试访问对应锁，进而持有它，直到完成结束或需要等待某项条件。当一个进程结束时，它将释放对应锁，从而不会发生死锁。

管程也可以拥有**条件变量**（`condition variables`）。为此，在相关条件不适合某线程（译者注：不妨设为线程 A）继续使用管程执行时，便可允许该线程 A 进行等待。在这种情况下，对应线程 A 将被阻塞，而另一个线程（译者注：不妨设为线程 B）将被赋予锁和允许执行。这里的另一个线程 B 可能会改变该管程的状态。如果现在相关条件满足正在等待的线程 A 继续执行的要求，那么正在运行的线程 B 可以向等待的线程 A 发出通知信号。这样将会把等待的线程 A 移回到就绪队列，以便其可以在管程变为空闲的时候恢复执行。以下代码使用了条件变量，用于实现每次只能存储一条消息的通信信道：

```
monitor channel {
    condition can_send
    condition can_receive
    char contents
    boolean full := false

    function send (char message) {
        while full then wait (can_receive)
        contents := message
        full := true
        signal (can_send)
    }

    function receive () {
        var char received
        while not full then wait (can_send)
        received := contents
        full := false
        signal (can_receive)
        return received
    }
}
```

9.4 死锁

9.4.1 什么是死锁

一个非常简单的例子

假设我们有两个进程 A 和 B，它们要共享两个不同的资源 1 和 2。进程 A 首先锁定资源 1，然后再锁定资源 2，接下来利用资源完成相关工作，继而再释放有关资源。进程 B 则首先锁定资源 2，然后再锁定资源 1，接下来利用资源完成相关工作，继而再释放有关资源。这些事件发生的粗略过程如图 9-3 所示。

197

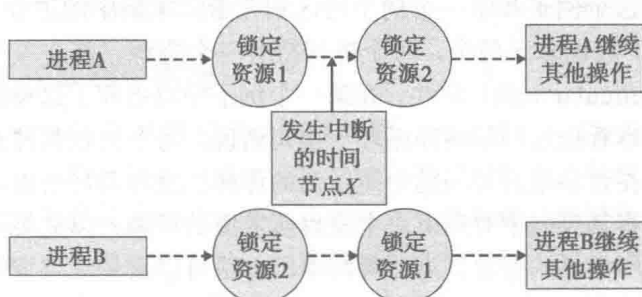


图 9-3 共享两个资源的两个进程

现在考虑一下，如果进程 A 在时间节点 X 处发生中断（有可能是进程 A 已经耗尽了它分配获得的时间片，故而操作系统将其从运行状态中取出并放回到就绪队列里），将会发生什么？这时候，进程 A 已经锁定了资源 1。不妨设想，现在进程 B 启动执行了，它先锁定了资源 2，然后便去尝试锁定资源 1。但是，由于进程 A 已经锁定了资源 1 并保持占有，所以操作系统就把进程 B 置为等待状态，之后还启动执行了其他某个进程。总有某个时刻，进程 A 会到达就绪队列的队首，并被调度器再次启动运行。它运行很短暂的时间便会尝试去锁定资源 2。然而，由于进程 B 已经锁定了资源 2 并保持占有，所以进程 A 也将被置为等待状态。于是，这两个进程现在就陷入了死锁状态。换句话说，这两个进程没有任何一个进程会完成，因为每个进程都持有另一个进程正在等待的资源。

从这个简单的例子，很容易可以看出死锁发生的两个必要条件。第一个条件是，必须涉及了不可共享的资源，称之为**互斥条件**（mutual exclusion）。就锁而言，这是非常清楚的，从其定义就可一目了然地看出——任何时间只有一个进程可以持有一个锁。但就某些资源来讲，这就不那么清晰了，后面将会就此进行讨论。死锁的第二个必要条件是，在进程等待一个资源的同时，其必须占有和保持另一个资源，称之为**持有 - 等待**（hold-and-wait）条件。再次地，就锁而言，我们可以看到，一个进程通常情况下可以获得很多它所需要的锁而不用释放它当前持有的任何锁。

一些较为复杂的例子

“哲学家就餐”问题是计算机科学文献中特别喜欢的一个例子。在这个问题里（如图 9-4 所示），有三位哲学家围坐在一张桌子旁边，或者吃饭，或者思考。一般来说，经过一段时间的思考之后，哲学家就想要吃饭。正在供应的膳食是米饭，且要求用两根筷子来进餐，而在桌子上每个对应两个

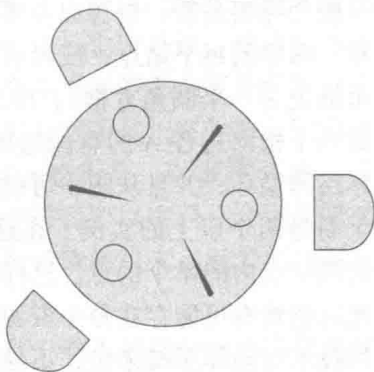


图 9-4 “哲学家就餐”问题

哲学家之间的位置都放有一根筷子（译者注：故而总共有三根筷子）。当要吃饭的时候，各位哲学家都是，先拿起一根左边的筷子，再拿起一根右边的筷子，然后开始吃饭。很明显，这种安排很容易会导致死锁。具体而言，设想某个时刻，三位哲学家先后决定了要吃饭，于是，每位哲学家都把手伸向了自己的左边，并各自拿起了一根筷子。然后，所有哲学家都被中断和停了下来，不得不要等上一会儿（就像上面简单的例子所描述的那样）。

当某位哲学家恢复处理（即继续试图取筷吃饭的时候），其尝试去拿起自己右边的筷子，却发现相应的筷子已经被占用了，所以只好停下来等待右边的筷子。我们让在桌子周围的这三位哲学家都尝试上一遍，最终他们也陷入了死锁状态——每位哲学家都持有一根筷子同时在等待另一根筷子。这个例子和第一个例子的区别在于，本例中的进程（哲学家）数要多于两个，而资源（筷子）数也多于两个。每个进程持有一个资源，并在等待另一个资源。这种情况称为循环等待（circular wait）条件。在第一个例子中也出现了这样的情况，不过由于只有两个进程，所以很难看出这种圆圈即循环。换句话说，每个进程都拥有一个资源是另一个进程所需要的。正如在哲学家就餐问题中所看到的那样，这种条件所要求的是，存在某种进程序列，每个进程持有其序列中对应下一个进程所需要的资源，以此类推，故而序列末尾的进程持有第一个进程所需要的资源。有一种简单的方法可以避免这类情形，我们将在本章后面的部分展开讨论。

在现实世界中经常引用的死锁的例子是城市街道上的格锁式交通堵塞。例如，图 9-5 就给出了一个简单的格锁式交通堵塞。（为简便起见，我们演示的是单向街道。）在这个例子里，你会看到许多不同的进程（汽车），每个进程都想使用其前面的那个进程已经在使用的资源。本例中，相应资源是指对应街道上的位置。很清楚，其间存在互斥——没有两辆车可以同时位于相同的位置上。同时还有循环等待——从图上看这是显而易见的。但是，请注意标记为 A 的那辆车。虽然这辆车也在等待，但它并不是死锁的组成部分，因为没有其他车正在等待它所持有的资源。

在关于死锁的许多分析中，还陈述了第四个条件——不允许抢占。抢占（preemption）意味着我们可以从某个进程那里拿走其当前持有的某个资源，故而会打破僵局、解除死锁。但是在我们的分析中，抢占却是死锁的一种解决方案。加上一项不允许抢占的“条件”仅仅是关于“死锁问题的一种可能的解决方案未被使用”的一种说法而已。实际上，这并不是死锁的必要条件。我们将在本章的后续部分就此展开进一步的讨论。

资源分配图

经常用来解释死锁的一种工具称为资源分配图（resource-allocation graph）。资源分配图一般会标明进程、资源以及哪个进程正在等待或者持有哪种资源实例。有关示例如图 9-6 所示。图中的每个结点，或者表示一个进程（这里标记为一个三角形），或者表示一个资源（这里标记为一个圆角方框）。由进程 B 指向资源 2 的有向边表示进程 B 正在等待资源 2，而由资源 1 指向进程 A 的有向边则表示进程 A 持有资源 1。如果存在死锁，那么图中将会存在环路即循环，并且从图中可以很明显地看出来。在计算机系统中，一个资源往往存在不止一个而是两个以上的实例。在这种情况下，传统做法是在图中把资源的每个实例标记为相应资源结点内的单个的点。这样，此类图中的一个环路或循环就不一定意味着存在死锁，因为此时仍然有可能存在每个资源的空闲可用的实例。遗憾的是，操作系统无法识别图，所以这种技术对操作系统来讲，不像对人类分析人员那么有用。诚然，程序设计人员可以模拟一幅图，并编写一个程序来完成心目中的对一幅图的搜索，但那不是一回事。

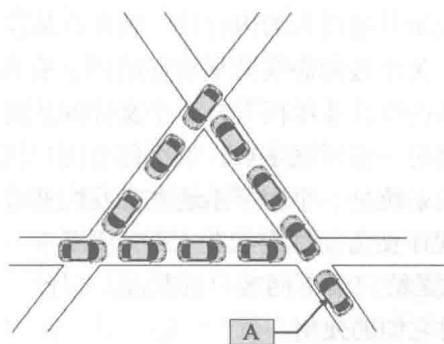


图 9-5 城市交通中的死锁问题

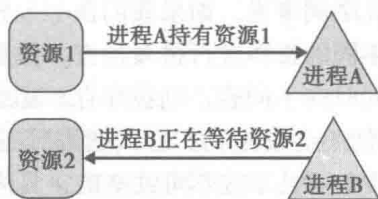


图 9-6 资源分配图示例

200

9.4.2 可以对死锁采取哪些措施

总的来说，有4种方法可以用来应对死锁问题。第一种，我们可以通过确保死锁发生的某项必要条件不成立来预防死锁的发生。第二种，我们可以允许所有这三项必要条件都发生，但是通过确保我们不会以死锁可能发生的方式来分配资源从而避免死锁。第三种，我们甚至可以允许死锁发生，检测死锁的发生，然后还可能针对发生的死锁采取某种措施。最后一种，我们可以直接忽略死锁和不予理睬。

9.4.3 死锁预防

预防死锁应当保证死锁的三项必要条件中的一项或多项不会发生。接下来，我们针对这三项条件依次阐明相应的死锁预防方案。

互斥条件

在计算机系统中，有些资源非常明显是不可共享的。举例来说，如果一个进程正在一台打印机上打印工资单，那么对于要在同一台打印机上开始打印电子邮件消息的另一个进程来说，将是行不通的，即不能执行的。（当然，我们可以通过假脱机输出技术来模拟对打印机的同时访问。我们将就此在本节的后续部分展开进一步的讨论。）类似地，如果一个进程正在向一台磁带驱动器写入记录，那么对于要开始使用同一台磁带驱动器的另一个进程来说，也是不可行的。有些其他的资源则明显是可以共享的。例如，网络接口卡（network interface card，简称网卡；或称为网络适配器，即 network adapter）则极有可能允许被多个应用程序同时共享。进一步说，一台服务器可能通过同一个网络适配器运行若干不同的服务——可能包括万维网服务器（Web server）、文件服务器（file server）和文件传输服务器（FTP server）。请求操作可能是从网络中的其他主机随机到达的，而响应则可能是由对应的服务器进程按照排队方式进行处理。有人可能会认为有关消息不是一起出去的——也就是说，相应线路并不是真正地“同时”使用的。然而，关键在于从来没有进程会不得不停下来等待网络发送数据。假设有足够的内存空间可以用作缓冲区，那么将没有进程会因为其正在等待向网络发送数据而陷入死锁。（诚然，有关进程可能必须得等待响应，然而这不是一码事。）同样，访问一台磁盘驱动器上的文件在软件级别上也是可以共享的。两个进程可以在同一台硬盘驱动器上打开有关文件，并且可以在单个的驱动器上对这些文件进行读取和写入，而无须等待其他的进程完全完成其相应的文件处理。

但有些资源则不是那么明显，例如随机访问存储器，即内存。有人可能会认为内存是可以共享的，因为许多进程可以同时使用内存的各个部分。然而，有关进程通常被赋予对内存

[201]

块的排他性访问权限，并且不允许其去访问被分配给其他进程的内存块。因此，从这个意义上讲，内存并不是真正可以共享的。尽管如此，关于进程确实共享内存的例子很多，所以从这点上来说，很难对内存进行归类。对于大多数的操作系统而言，单个文件的访问可以是共享的。举例来说，如果我们在一个分时系统上拥有一部拼法字典，那么每个用户就可以同时针对不同的文档进行拼写检查。但是，如果相应系统是一个库存系统并且我们拥有若干进程试图同时向不同客户划拨库存，那么有关应用程序便需要锁定相关文件（或至少一部分文件），以便不会发生我们尝试把最后一个小部件发送给三个不同客户的状况。因此，文件并不是本质上可共享或不可共享的，具体应取决于对它们的使用。

即使是对于如打印机之类的不可共享的设备，我们也可以利用某些机制把针对一台打印机的绝大部分多数情况的使用转变成为一种可共享的事件。其对应的解决方案就是使用假脱机技术。操作系统不是直接将数据写入打印机，而是将从应用程序获取到的数据临时存放在一个磁盘文件中。随后，当它知道对应打印机可以使用且安装了合适的打印纸等前提的情况下，才会真正把数据打印到打印机上。因为我们已经清除了涉及打印机的互斥条件，所以我们就把打印机移除出可能引发死锁的资源列表。另一方面，即便是使用假脱机技术也可能会发生某种类型的死锁。进一步说，当操作系统为多个应用程序假脱机打印机输出时，它会暂时把相关输出写到磁盘上，而分配给假脱机模块的磁盘空间被填满是完全有可能的。因此，这便可能再次让系统暴露在可能的死锁状态。

然而，至关重要的问题是，由于某些资源本质上是不可共享的，所以消除互斥条件并不是一种普遍适用的解决方案。

持有 - 等待条件

我们可以采用两种途径来避免死锁发生所必需的持有 - 等待条件。首先，我们可以要求一个进程必须在其启动时就请求它在整个执行过程中所需要的所有的资源。对于一些简单的批处理系统来说，这有可能是可以接受的，但是对于大多数现代的应用系统来说，则是不可行的。仅仅从关于可能事件的太多的组合这一点上来说，对所有需求的预测就是无法实现的。此外，80/20 法则——也就是说，在 80% 的情况下，我们仅仅需要很少量的资源；而只有在 20% 的情况下，我们才需要较大份额的其他内存——适用于大多数的情况。如果我们要按照最坏的情形来提前申请资源，那么大多数时候我们将会把我们并不需要的资源无效地占有起来而使得这些资源不能被用到其他真正需要和发挥作用的地方。

第二种方案是，要求申请资源的任何进程在申请任何其他资源之前必须首先将其持有的所有资源释放掉。于是，在我们的第一个例子中，当进程 B 想要申请资源 1 时，它必须首先释放资源 2，然后再同时申请资源 1 和资源 2。但因为进程 A 占有资源 1，所以进程 B 不能分配获得这两个资源，只好等待，而且它现在不再持有资源 2。某个时候，进程 A 终将获得其下一个时间片，进而首先释放资源 1 并尝试同时申请资源 1 和资源 2。由于进程 A 当前占有处理器，所以它将被允许锁定这两个资源，并将继续执行。当它利用这两个资源完成相关任务并释放掉它们之后，某个时候进程 B 也将最终被置于就绪状态，并且将被授权访问这两个资源和继续执行。这样，我们便防止了死锁。然而，如果一个应用程序使用的是不可共享的资源，有关应用程序如何释放相应资源呢？还有，这种不断地释放和重新锁定除了在最简单的情况下可以使用之外，其效率也未免有些太过低下了。因此，就像互斥条件一样，消除持有 - 等待条件通常并不是解决死锁问题的一种有效方案。

循环等待条件

死锁的最后一个必要条件是循环等待，而且有一种非常简单的通过不允许这种情况发生来预防死锁的方法。具体而言，相应的解决方案要求首先针对系统中的所有资源确立一种排序。（除了当有关排序与相关程序最有可能锁定资源的顺序相匹配的情况下可以取得最好的运行效果之外，这种排序并没有任何实际意义。）在此基础上，要求所有的进程都必须以相同的次序来锁定资源。这样便可以防止循环等待条件的形成。再次考虑第一个例子。如果进程 A 和进程 B 均在尝试锁定资源 2 之前都先尝试锁定资源 1，那么就永远不会出现这两个进程之间的死锁。因为当第二个进程尝试锁定资源 1 时，它将被强制等待。如果牵涉多个进程和多个资源，这种解决方案也很有效。

202

遗憾的是，作为操作系统中用于死锁预防的一种通用的解决方案，资源排序并不是一种真正可行的解决方案。一方面，操作系统实用程序、第三方软件和终端用户应用程序等各方都不得不按照在内心的某种这样的标准来编写，另一方面并不存在这样的标准。尽管如此，对于在具有多个并发运行子系统的大型系统上工作的开发团队来说，资源锁定排序则是避免在相关应用系统自身内部产生死锁的一种有效技术。因此，即便这并不是死锁问题的一般的解决方案，但它却是一项需要掌握的重要技术。

9.4.4 死锁避免

到目前为止，我们关于资源的示例中，往往只说明了每种资源的一个实例。一个简单的锁每次只能有一个用户，一台打印机每次也只能有一个用户，以此类推。但对于其他的资源来说，一般可以有相应资源的许多实例。最为明显的例子是随机访问存储器（即内存）——也就是说，总会有许多内存块要分配。类似地，我们可能拥有多台可以用来安装磁带的磁带驱动器。在一台大型机上，我们甚至可能拥有多台相同的打印机，而不用真正关心我们到底要使用哪一台。在研究死锁避免机制时，我们常常要考虑相关资源拥有多个实例这样的更为一般化的情况。

对于死锁避免而言，通常有两种机制，而且这两种机制均要求：在一个进程运行之前，它必须向操作系统提供其将会申请的每类资源实例的最大数量（涵盖所有各种情况）。举例来说，它可能会声称，自己将仅仅需要 543KB 的内存、一台打印机和三台磁带机。然后，操作系统可以采用两种方式来使用这些数据。第一种方式是使用有关数据来决定是否要运行相应作业。进一步说，当操作系统要启动一道作业时，操作系统可以查看系统当前可用的各类资源数量，并确认这些资源是否可以满足应用程序可能申请的最大需求数量。不妨假设，系统现在可能拥有打印机和三台磁带驱动器可以分配给对应程序，但却只有 506KB 的内存。如果操作系统无法确保能否授予对应作业可能请求的所有各类资源的最大数量，那么它将不会运行对应作业。这样，操作系统就会避免将自己置于可能发生死锁的境地。这当然是非常安全的，但却不是一种充分最优化的解决方案，因为有关作业可能常常在不需要获得其最坏情况下所需各类资源的条件下运行。这其实等效于要求进程提前申请所需的所有资源。

第二种解决方案要更困难些，但更接近于最优化。对于这种方案而言，操作系统将不用检查作业所需的最大资源数量就可以启动作业的运行，但是当程序请求任何资源时，操作系统将需要确定自己能否满足该项申请以及分配后是否仍然能够完成所有已经在运行的各道作业。如果系统不能安全地满足相应进程所提出的资源申请，那么它将把该进程置为等待状态。即使所有当前正在运行的作业都按照它们所声称的可能使用的所有资源最大数量来

203

进行申请，操作系统也可以断定自己可以顺利完成所有这些作业运行的状态被称为安全状态 (safe state)。

在图 9-7 所给出的示例中，操作系统正在监控 A、B、C、D 四类资源。对于这些资源来讲，系统当前分别拥有 1、5、2、0 个未分配的（可用的）实例。需要说明的是，为了简便起见，我们在图中显示这些资源的计数列表时，没有带逗号。其中，有 0、1、2、3、4 共 5 个进程。当这些进程开始运行时，它们均都给出了自己关于四类资源所可能申请的各种资源实例的最大数量，具体由图中标题为 Max 的列所给出。举例来说，进程 1 声称自己最多需要 1 个 A 类资源实例、7 个 B 类资源实例和 5 个 C 类资源实例，且不需要 D 类资源实例。每个进程当前所持有的每种资源实例的数量，具体由图中标题为 Alloc 的列所给出。正如我们可以看到的，进程 2 当前已经分配占有了 1 个 A 类资源实例、3 个 B 类资源实例、5 个 C 类资源实例和 4 个 D 类资源实例。进一步说，操作系统可以确定进程 1 不会再请求任何 A 类资源实例，因为它已经分配获得了它所声称最大需要的那么多的相应实例。如果进程 1 还要申请更多的 A 类资源实例，我们可以终止这一进程。当然，进程 1 还可以再申请 7 个 B 类资源实例和 5 个 C 类资源实例，不过不能再申请 D 类资源实例。这些关于进程运行尚需申请和分配的资源数量具体由图中标题为 Need 的列所给出。操作系统可以根据已有的这些数据和信息检查其是否可以顺利地完成这批作业，而不会发生死锁。具体来说，我们注意到，在当前情况下是无法断定进程 1 能不能完成的，因为它还要求另外的 7 个 B 类资源实例，而我们现在只有 5 个 B 类资源实例。但是，我们可以断定进程 0 将能够顺利完成，因为该进程再没有其他的资源要求。当进程 0 完成结束后，我们可以回收原先分配给该进程的资源——在这里即为 0 0 1 2。于是，系统现有可用各类资源实例的数量就变成了 1 5 3 2。关于推演过程中各类资源的可用数量具体由图中标题为 Work 的列所给出。到目前为止，仍然无法断定进程 1 能否顺利完成，但是我们可以确定进程 2 可以顺利完成了，因为它尚需申请的各类资源的数量（即 1 0 0 2）都比我们现在所拥有的可用资源数量（即 1 5 3 2）少。当进程 2 完成结束后，我们也将回收其原先分配占有的资源（即 1 3 5 4），于是我们拥有的可用资源数量变为 2 8 8 6。现在，我们终于可以确定进程 1 能够顺利完成了，于是我们所拥有的可用资源数量就变成了 3 8 8 6。类似地，在现有资源状况下，我们可以确定进程 3 和进程 4 能够顺利完成。鉴于我们可以确认所有的进程都能够顺利地完成，所以我们可以断定，系统当前处于安全状态。

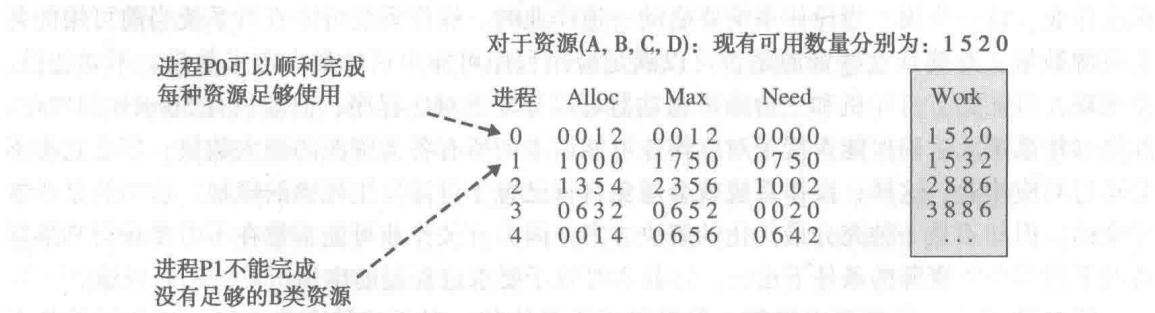


图 9-7 安全状态说明示例

因此，为了避免死锁，操作系统必须通过检查每个进程关于资源分配的每次请求，从而确保如果系统满足和同意相应的资源申请，系统仍将处于安全状态。不过请注意，一个不安全的状态并不意味着我们陷入了死锁或者我们肯定会陷入死锁，这仅仅意味着我们可能最

终会陷入死锁。可以确定无疑的是，通过始终不允许系统进入不安全状态，我们将会避免死锁。尽管如此，我们将再次以一种次优的方式来使用系统，因为我们可能会让有关进程在可能成功运行和没有死锁的情况下等待。我们刚刚非形式化描述的这种算法被称为银行家算法（Banker's Algorithm），它曾经被一个称为 THE 的操作系统（译者注：THE 为“Technische Hogeschool Eindhoven”的缩写，即荷兰语中的埃因霍温科技大学）使用过。然而，对于许多系统来说，要预先知道每个进程将会申请哪些资源和最多申请多少是不可能的，因此死锁避免方法在当前的操作系统中并没有实际应用。

204

9.4.5 死锁检测

我们解决死锁问题的另一种方法是，让死锁发生，并且当发生死锁时，检测到死锁的发生和尝试解除死锁、恢复系统正常状态。这种方法的主要优势在于，它会让所有的进程尝试运行，并且从来不会仅仅为了避免可能的死锁就使有关进程停下来等待。从这个意义上来说，这种方法是最佳的。

如果我们确实想要就涉及每类资源多个实例的资源分配冲突进行检测，我们可以利用与银行家算法类似的一种算法。就该算法来说，我们将会审视当前申请但尚未分配的资源量，而不是尚未申请的最大资源量。在这种情况下，如果我们不能找到一个“安全状态”，那么我们将会认识到相关无法完成的进程可能卷入了某种死锁。然而，现在没有任何实际的操作系统包含了这样的算法。相反，操作系统把此类算法留给了那些担心死锁的应用程序，因为这些都是在实际会遇到的死锁类型。有关的操作系统则提供了一个应用程序接口调用，允许相关应用程序对所有等待任务的列表进行检查。这样，相关应用程序就可以检查所有等待任务并确认它们中间是否存在循环。事实上，这种检查一般是由运行用户应用程序的调试程序来完成的。如果对应调试程序发现了一个循环，那么程序员就可以检查有关数据并解决相关问题。

9.4.6 抢占和其他实用的解决方案

某些资源是按照这样的方式来使用的，一旦某个进程开始使用它们，那么对应进程就需要完成其正在完成的事情，之后我们才能够使用这些资源。比较不错的例子如，把文件写入磁带驱动器或者输出到（没有假脱机技术支持的）打印机。其他的资源则有所不同，如随机访问存储器，即内存。进一步说，如果有两个进程正在运行，并且每个进程都需要比系统可以为其提供的内存量还要多的内存，那么我们可以暂时挂起其中的一个进程，将该进程当前在内存中的所有信息保存到辅助存储器上，然后让另一个进程拥有所有内存。这样，当第二个进程完成结束后，我们再将第一个进程恢复加载到内存，分配给其想要的额外的内存，并让其继续运行。这种技术被称为抢占（preemption）。再比如说，在图 9-5 中，我们就可以应用抢占技术——让执勤警察要求图中左下角的汽车司机往后退，即抢占该汽车原先在街道上的位置——来解决格锁式交通阻塞。

那么接下来的问题是哪些作业应当被抢占。通常情况下，最好的选择是具有最小成本的那道作业——比如说，当前拥有足够大的满足当前申请需要的但却是最小内存量的那道作业。而如果抢占了最大资源量的进程还没有释放出足够的资源使剩余的作业完成，那么就可能需要对下一个比最大资源量小一点资源量的进程来重复这一抢占处理措施。

但是，如果死锁牵涉的所有进程正在等待的是无法被抢占的资源，那么我们可能别无选

205

择,而只能中止某些或所有相关进程。需要说明的是,通常的选择一般是中止对应死锁相关的所有进程,这似乎有点不同寻常。然而,死锁往往是一个非常罕见的事件——这种事件是如此罕见,故而使其可能并不值得花时间去开发比较复杂的算法。此外,可用来开发这类算法的基础数据是非常稀少贫乏的。一种更好的选择是持续不断地尝试中止最低成本的那个进程,直到对应死锁消失为止。当然,在每次尝试之后都应当再次运行死锁检测算法。

如前所述,最常用的解决方案是忽略死锁问题。遗憾的是,死锁所涉及的进程往往可能会消耗大量的资源。由于最初陷入死锁的相关进程所保持的死锁,其他的进程也可能最终会停下来加入等待的行列。从而到最后,系统可能停止运行任何进程。希望系统操作员将会注意到有关问题,并开始着手解决问题(可能是通过中止作业的方式),直到系统恢复正常运行。

在未来,似乎操作系统将会包含更多的机制来应对死锁。虽然用于死锁检测的算法确实会需要和消耗一些处理器和内存资源,但是死锁对于用户来说却是非常诡秘的——他们正在使用的系统看起来只是悬在那里,而他们竟然无能为力,既不知道该如何解决此类问题,也不知道如何在将来避免此类问题。与此同时,计算机硬件不断地获得更为强大但又更为低廉的内存。目前,死锁检测正在针对有关调试程序加以实现,而且我们推测相关调试程序在将来定会找到它们作为后台功能而进入内核的途径。

9.5 小结

在本章中,我们讨论了由多个协作式进程所组成的系统的相关机理。我们从分析为什么系统经常以这种方式来进行构建而且有关趋势似乎正在不断增强的原因出发来开启本章的介绍。我们审视了进程用来实现彼此相互通信的有关机制。然后,我们研究了当两个进程之间每个进程都试图去访问另一个进程正在(或可能在)同时访问的数据时所出现的问题。我们阐述了已经开发出来用于允许进程同步它们的活动从而避免相关问题的一些工具。最后,我们讨论了另一类问题,即当多个进程使用同步机制来锁定资源时可能出现的所谓死锁问题。我们描述了防止死锁把我们的系统带向停止状态的4种理论上的相关机制。

在下一章中,我们将介绍基本的系统内存的管理。

参考文献

- Ben-Ari, M., *Principles of Concurrent Programming*. Englewood Cliffs, NJ: Prentice Hall, 1982.
- Bernstein, A. J., "Output Guards and Nondeterminism in Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2, 1980, pp. 234-238.
- Brinch Hansen, P., "Structured Multiprogramming," *Communications of the ACM*, Vol. 15, No. 7, July 1972, pp. 574-578.
- Brinch Hansen, P., *Operating Systems Principles*. Englewood Cliffs, NJ: Prentice Hall, 1973.
- Coffman, E. G., Jr., M. J. Elphick, and A. Shoshani, "System Deadlocks," *Computing Surveys*, Vol. 3, No. 2, June 1971, pp. 67-78.
- Courtois, P. J., F. Heymans, and D. L. Parnas, "Concurrent Control with Readers and Writers," *Communications of the ACM*, Vol. 14, No. 10, October 1971, pp. 667-668.
- Dijkstra, E. W., "Co-operating Sequential Processes," in F. Genuys (Ed.), *Programming Languages*. London: Academic Press, 1965, pp. 43-112.
- Dijkstra, E. W. *EWD 126: The Multiprogramming System for the EL X8 THE* (manuscript), 14 June 1965.
- Dijkstra, E. W., "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM*, Vol. 8, No. 5, September 1965, p. 569.
- Dijkstra, E. W., "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, Vol. 1, 1971, pp. 115-138.
- Eisenberg, M. A., and M. R. McGuire, "Further Comments on Dijkstra's Concurrent Programming Control Problem," *Communications of the ACM*, Vol. 15, No. 11, November 1972, p. 999.
- Habermann, A. N., "Prevention of System Deadlocks," *Communications of the ACM*, Vol. 12, No. 7, July 1969, pp. 373-377, 385.

206

- Havender, J. W., "Avoiding Deadlock in Multitasking Systems," *IBM Systems Journal*, Vol. 7, No. 2, 1968, pp. 74–84.
- Hoare, C. A. R., "Towards a Theory of Parallel Programming," in C. A. R. Hoare (Ed.), *Operating Systems Techniques*. New York: Academic Press, 1972, pp. 61–71.
- Holt, R. C., "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys*, Vol. 4, No. 3, September 1972, pp. 179–196.
- Howard, J. H., "Mixed Solutions for the Deadlock Problem," *Communications of the ACM*, Vol. 16, No. 7, July 1973, pp. 427–430.
- Isloor, S. S., and T. A. Marsland, "The Deadlock Problem: An Overview," *IEEE Computer*, Vol. 13, No. 9, September 1980, pp. 58–78.
- Kessels, J. L. W., "An Alternative to Event Queues for Synchronization in Monitors," *Communications of the ACM*, Vol. 20, No. 7, July 1977, pp. 500–503.
- Knuth, D., "Additional Comments on a Problem in Concurrent Programming Control," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 321–322.
- Lamport, L., "A New Solution to Dijkstra's Concurrent Programming Problem," *Communications of the ACM*, Vol. 17, No. 8, August 1974, pp. 453–455.
- Lamport, L., "Synchronization of Independent Processes," *Acta Informatica*, Vol. 7, No. 1, 1976, pp. 15–34.
- Lamport, L., "Concurrent Reading and Writing," *Communications of the ACM*, Vol. 20, No. 11, November 1977, pp. 806–811.
- Lamport, L., "The Mutual Exclusion Problem: Part I—A Theory of Interprocess Communication," *Journal of the ACM*, Vol. 33, No. 2, 1986, pp. 313–326.
- Lamport, L., "The Mutual Exclusion Problem: Part II—Statement and Solutions," *Journal of the ACM*, Vol. 33, No. 2, 1986, pp. 327–348.
- Lampson, B. W., and D. D. Redell, "Experience with Processes and Monitors in MESA," *Communications of the ACM*, Vol. 23, No. 2, February 1980, pp. 105–117.
- Levine, G. N., "Defining Deadlock," *Operating Systems Review*, Vol. 37, No. 1, pp. 54–64.
- Newton, G., "Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography," *ACM Operating Systems Review*, Vol. 13, No. 2, April 1979, pp. 33–44.
- Patil, S. S., "Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination among Processes," M.I.T. Project MAC Computation Structures Group Memo 57, February 1971.
- Peterson, G. L., "Myths About the Mutual Exclusion Problem," *Information Processing Letters*, Vol. 12, No. 3, June 1981, pp. 115–116.
- Raynal, M., *Algorithms for Mutual Exclusion*. Cambridge, MA: MIT Press, 1986.
- Zobel, D., "The Deadlock Problem: A Classifying Bibliography," *Operating Systems Review*, Vol. 17, No. 4, October 1983, pp. 6–16.

网上资源

<http://boinc.berkeley.edu> (SETI and BOINC)
http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html
<http://www.softpedia.com/get/Others/Home-Education/>

Deadlock-Avoidance-Simulation.shtml
<http://webscripts.softpedia.com/script/Development-Scripts-js/Complete-applications/Banker-s-Algorithm-Demonstration-15119.html>

207

习题

- 9.1 关于为什么有时希望把一个系统分解成不同的进程，有时运行在不同的机器上，我们列出了如下8种原因。对于其中的每一种原因，请给出一个与课本中所给例子不同的另外的例子。
- 性能——
 - 规模扩展——
 - 已购买的部件——
 - 第三方服务——
 - 多系统中的组件——
 - 可靠性——
 - 信息的物理位置——
 - 志愿式启用型应用程序——
- 9.2 对于进程间通信机制的每项特性来说，一般相应的特性都会有各种各样的选择。请讨论如下特性的相关各种选择的优点和缺点：
- 可能有多个连接或单一连接——
 - 命名策略——
 - 面向连接的通信或无连接的通信——
 - 持久通信或瞬时通信——
 - 进程数——

- 9.3 管道是阻塞式通信机制的一个实例。这是否正确?
- 9.4 套接字是持久通信机制的一个实例。这是否正确?
- 9.5 采用共享内存进程间通信机制的最大问题是什么?
- 9.6 为什么同步问题很难调试?
- 9.7 关于我们所讨论过的用于同步的所有硬件锁定指令来说,其特别之处是什么?
- 9.8 为什么应用程序不使用自旋锁?它们使用的是什麼?
- 9.9 关于上锁和开锁的系统调用,其相应的标准的名字是什么?
 - a. lock 和 unlock
 - b. set 和 clear
 - c. wait 和 signal
 - d. enter 和 exit
 - e. 以上均不是
- 9.10 有些特殊的信号量称为计数型信号量。它们是用来完成什么事情的?
- 9.11 当运行在对称多处理系统上时,应用程序必须采取特殊的预防措施来确保任何锁的取值可被所有的处理器看到。这是否正确?
- 9.12 简要描述优先级倒置的概念。
- 9.13 什么引发了高级语言中管程的开发?
- 9.14 死锁发生需要三个条件。它们是什么?
- 9.15 我们说过,对于死锁的发生而言,存在一个进程序列,其中每一个进程持有一个资源,同时还在等待着下一个进程所持有的另一个资源,且最后一个进程等待着第一个进程所持有的资源。试问多少个进程创造了一个死锁?
- 9.16 有些设备是不可共享的,但是我们找到了一种构建虚拟设备的方法,允许我们假装成是在共享相关设备。那种机制是什么?
- 9.17 关于各种资源所对应的锁的排序能够消除循环等待,从而消除死锁。什么时候这种技术可以使用?而什么时候这种技术不可以使用?
- 9.18 我们讨论过两种不同类型的死锁避免。一般而言,死锁避免存在什么问题呢?
- 9.19 关于死锁检测的算法众所周知且不难编写,但为什么我们不经常使用呢?
- 9.20 方便“抢占”方法使用且效果很好的例子是什么?

可能要求拓展阅读的问题:

- 9.21 现代操作系统出于不同的目的而使用了若干不同类型的信号量。挑选一种现代的操作系统,说出其所支持的一些不同类型的信号量的名字,并分别给以简要的解释说明。
- 9.22 有许多牵涉同步的经典问题,我们已经描述了两个。你能找出我们描述了哪两个经典同步问题吗?而其他的经典同步问题又有哪些呢?

基本的内存管理

10.1 为什么要管理主内存

在前面的两章中，我们讨论了操作系统必须要完成的一项主要工作，即管理在处理器上运行的进程。在本章中，我们将讨论操作系统的第二项主要工作，即管理主内存（primary memory，简称主存或内存）。与所有系统资源一样，操作系统尝试管理系统的主存。通常情况下，主存是由电子电路制成的随机访问存储器（Random Access Memory，RAM）^①。内存管理的基本目标是允许尽可能多的进程运行。操作系统提供应用程序接口函数，以允许我们分配、访问和回收内存。当内存被分配给进程时，操作系统必须监测记录内存的哪些部分是空闲的、哪些部分是被分配的以及相关部分被分配给了哪些进程。操作系统有时也可能从进程那里抢占内存。在这种情况下，它必须准备在辅助存储器（secondary memory）上保存主存的当前内容，记录每个进程的各部分内容的存储位置，以及当被抢占的进程恢复时恢复主存的内容。

209

在大多数情况下，操作系统将尝试以对应用程序透明的方式来管理内存。然而，我们后面在下一章中将会注意到，在某些情况下，这种透明是不彻底的。对于试图优化性能的大型系统来说，关于内存服务的幼稚使用有时可能会导致相关问题的发生。事实上，我们学习操作系统的一个主要理由就是要获得必要的信息和理解能力，以克服这些问题。

在讨论了为什么我们要管理内存之后，我们在第 10.2 节中，将会阐明一个进程的传统开发和运行周期模型，以及把一道程序中的某一项的引用绑定到存储该项的物理内存位置的相关步骤。而我们后面将会使用这些步骤来解释各种各样的内存管理机制。其后，我们将从第 10.3 节中的单一进程开始，逐渐展开讨论在越来越复杂的情况下的内存管理，包括讨论诸如重定位和覆盖等各个方面。接下来，我们转向牵涉多个进程的情况，再次渐进式地讨论越来越复杂的相关机制，包括在第 10.4 节中的针对固定数量的进程的管理以及在第 10.5 节中的针对可变数量的进程的管理。最后，我们对全章内容进行了归纳总结。

10.2 开发周期步骤与绑定模型

首先，让我们描述构建一个应用程序，将其加载到内存并予以运行的相关步骤的标准模型。在本章的后续各节中，我们将使用这个模型围绕如何管理进程使用内存的方式来解释操作系统的相关常见功能。

实际上，以一个进程在内存中运行至结局的相关事件序列共由 5 个步骤组成。第一步，我们编写（或编码）对应程序。通常这是利用某种符号语言来完成的，或者是低级的汇

① 系统有可能是使用其他类型的主存构建的。例如，早期的系统就曾使用了水银槽中的声波或旋转的磁鼓。曾有一些年，几乎所有的计算机系统都使用了氧化铁芯中的磁化极性来存储位。短语“核心转储”——对分配给一个已经崩溃的程序的内存的内容的打印输出——就源自这个时代。这种内存具有即使在电源关闭时仍然保留其内容的特性。今天，这作为一项特有的属性吸引着我们，因为我们通常认为主存是易失性的。

编语言，或者是高级的面向问题的语言。第二步，我们使用一种翻译程序，通常是汇编程序或编译程序，但偶尔也会是解释程序（译者注：此三者对应英文术语分别为 assembler、compiler、interpreter，或分别称为汇编器、编译器和解释器），将这个符号程序转换（translate）成目标机器的指令序列，创建一个“目标模块”。然而通常情况下，这个目标模块尚不能直接运行。第三步，我们一般会将该模块与各个单独创建的类似的模块进行链接（link）。这些模块可能是我们创建的其他模块，也可能是我们购买的或者与操作系统一起提供的库模块。第四步，我们将上一步链接形成的对应程序加载（load）到内存中。第五步，我们真正运行（run）这个程序。

请注意这些步骤的名称：在历史上曾经设计和出现过许多不同的软件包，用来辅助程序设计人员实现一个程序。因为不同的系统一般被用于解决不同环境中的不同问题，所以其中一些步骤的功能有时就被组合成某一单个的模型。我们所描述的把相关模块组合在一起的功能通常被称为链接（linking），而把有关进程带入内存的功能通常被称为加载（loading）。然而，有些时候，这些功能可能在一个步骤中就可完成。在其他的文献中，你可能会看到某一个词用来描述某一个步骤，或者是用来同时描述这两个步骤的组合。

假设我们正在创建由两个模块组成的一个进程。我们拥有一个主程序，其中调用了我们先前所编写的名为 XYZ 的子程序。在学习越来越复杂的模型时，绑定问题将是我们重点关注的。绑定（binding）就是指确定子程序应当放在物理内存的什么地方并使主程序中的有关引用指向相应子程序的过程。但是，对于并非主程序一部分的所有项的引用，都会发生绑定，也就是说这里的项不仅仅是子程序项，还包括数据项。

10.3 单一进程

10.3.1 编码时绑定

对于像具有非常小的基本输入/输出系统（BIOS）的嵌入式系统这样的非常简单的运行环境，我们可能通过手工方式就可以确定程序的每一部分会放到什么地方或位置。我们可能将主模块放在位置 100，而将子程序 XYZ 放在位置 500。如果我们采用汇编语言来编写主模块，那么我们可能包含像 ORG 100 这样的汇编指令，而在子程序中则可能包含一条汇编指令为 ORG 500。这些指令便会导致汇编器生成以绝对方式引用这些内存地址的相应代码。于是，在我们的主模块中，我们就会知道子程序 XYZ 将被放在地址 500，故而我们实际上就可以用针对位置 500 的调用请求来替代针对 XYZ 的调用请求。在这种情况下，我们在编码步骤便可完成绑定决策，并且是通过 ORG 指令把相关结果告诉了汇编器。今天，这对我们看来似乎不太可能，但这绝对算不上早期绑定的一个极端例子。下面说明三个极端的例子：

当计算机首次研制出来时，以及当小型计算机和个人计算机首次研制出来时，第一批系统拥有非常小的软件和很少的外围设备。程序员不仅仅是手工分配地址，而且还是在用机器语言编写程序，甚至是通过操作机器前面的控制板上的开关和按钮手动将有关程序输入存储器中。在某些机器中，还有使用固定内存地址作为缓冲区的外围设备，故而程序员甚至都别无选择。不用说，让程序员以手工方式来分配内存是容易出错且耗费时间的东西。这个阶段并没有持续很长的时间。

IBM 650 计算机系统拥有一个主存，且是一个旋转的磁鼓。有关指令的执行一般会花费

可变长度的时间。当然，在对应指令执行的过程中，磁鼓仍然在继续旋转。为此，每条指令往往会包含一个字段以用于给出下一条指令的地址。而程序员不得不尝试通过将下一条指令放置在当前指令完成时的磁鼓读取头下方的下一个位置来优化程序。显然，这一阶段也没有持续很长的时间。期间，在哥伦比亚大学开发了一个名为 SOAP (Symbolic Optimizing Assembler Program, 符号优化式汇编程序) 的汇编器，其主要工作就是实现相关指令在磁鼓上的优化放置。

在程序被例行公事地以打孔方式记录到卡片上并且没有磁带或旋转式存储器可用的时间里，将汇编程序加载到存储器、“喂进”源程序、获取以打孔方式记录到卡片上的目标模块、将链接程序 (linker program) 加载到存储器、将对应目标模块“喂给”链接程序、以打孔方式把可执行程序 (executable program) 记录到卡片上，以及最后将目标程序 (object program) 加载到计算机上加以运行，这是一个相当复杂的过程。因此，针对以打孔方式记录到卡片上的可执行程序进行修补，即打补丁 (patch)，是司空见惯的事情。鉴于有关的汇编清单包含有对应于每条指令而输出的机器语言，这便允许程序员找到包含不正确指令的卡片，将其加载到一个键控打孔机上，并通过直接更改对应的机器语言来修复程序。遗憾的是，这种做法确实持续了一段时间，而且也很容易出错。此外，由此导致程序员将会在一叠卡片中插入多个补丁，最终，补丁的数量将会变得非常笨拙和极不便利，所以程序员只好返回到源码卡片叠，针对源程序重复所有的更改处理，然后重新执行汇编等一系列过程。遗憾的是，漏掉针对源代码的一两处更正又是太过容易的事情，因此发现有人在源代码中修补他自己已经用补丁修复过的错误，这样的情况并不罕见。

[211]

10.3.2 链接时绑定

在 CP/M 运行环境中，所有程序都应该从位置 100 开始，因此在主模块中，可能包含 ORG 100 这样的语句。同时，我们可能并不关心子程序 XYZ 在哪里结束，所以在主例程中我们可以使用符号名称 XYZ。当汇编器输出目标模块时，其中包含所有的指令和数据，同时还包含某些信息用来告诉链接程序，即我们有一些引用需要其进行修复或链接。当链接程序处理主例程后，它将会拥有一张其需要解析的名称列表，然后它将开始处理我们先前告诉它的那些应包含的其他模块。伴随它在加载步骤中真正包含这些模块，它将在那些模块中找到在其中所定义的名称。在这个例子里，“XYZ”的定义将会出现在我们先前告诉链接程序进行处理的某个模块中。当链接程序计算和确定模块 XYZ 在地址空间中的位置时，它将返回去把主模块中的相关引用链接 (或绑定) 到对应子例程模块中的地址上。现在，我们是在链接时而不是编码时才完成了绑定决策。请注意，当我们把绑定推迟到后面的步骤时，我们获得了一定程度上的灵活性。如果我们在编码时就已决定把模块 XYZ 放在位置 500，而后来发现需要将该模块移动到其他位置时，我们将不得不花费很多精力来修正关于这一地址的所有的引用。但让链接程序来决定模块的存放位置将使得这一切变得非常方便和容易。不过话又说回来，为了灵活性的增强，我们还是付出了一些代价的。进一步说，在这种情况下，我们需要在目标模块中增加额外的信息用来定义相关名称以及有关名称的引用，并且我们在链接步骤中为其所做的绑定会花费一些额外的时间。尽管如此，随着计算机的运行速度变得更为迅捷而存储空间变得越来越大，需要付出的这种额外的时间和空间的开销已经变得微不足道，以至于我们很可能甚至都不会把它放在心上。

10.3.3 单一进程

在 CP/M 运行环境中，操作系统位于内存的顶端，而应用程序从位置 100 开始，以避免放置在低端内存中的中断向量（interrupt vector），并按向上方向增长。遗憾的是，随着操作系统变得越来越大（而且它总是如此），有可能最后变得很大，以至于升级后的操作系统可能使用了升级之前曾经运行良好的应用程序所需要的内存。于是，系统将会发生崩溃，并且有可能连什么问题都说不清楚。因此，微软 DOS 操作系统采取了一种不同的方法：操作系统被加载到低端内存，而应用程序加载到它上面的位置。当应用程序试图加载时，如果有足够的空闲可用内存，那么对应程序将进行加载并准予运行。但如果如果没有足够内存，那么至少对应故障已被明确判定。最初，当有人在微软 DOS 操作系统下创建应用程序时，对应链接生成的程序只能在特定版本的操作系统上运行。链接过程中确定了操作系统中的相关服务例程的地址以及应用程序应当开始加载的地址。遗憾的是，这便意味着，当常驻操作系统的绝对大小发生改变的新版微软 DOS 操作系统发布的情况下，所有的应用程序必须得重新链接之后才能在新版操作系统上运行。（译者注：其实，如果新版操作系统中相关服务例程地址发生变化，即使其常驻操作系统的绝对大小没有发生变化，有关应用程序也需重新链接。）

212

许多大型机操作系统具有类似的体系结构，但它们利用了额外的硬件来保护自身。在图 10-1 中，我们可以看到一个典型的早期操作系统的体系结构，那是只支持一个进程运行的一台大型机的体系结构。可执行程序将会被创建和驻留在操作系统内核上方的某一特定地址。此外，一个基址寄存器（base register）将会用某个地址加载设定，且可执行程序不能寻址低于该地址的空间。如果程序确实引用了这一地址以下的内存，就会产生一个中断并终止对应程序。这样的系统仍然会存在问题，因为如果操作系统变大，那么应用程序将不得不根据新的地址来进行链接。这一问题的解决方案是稍微改变一下基址寄存器的功能。

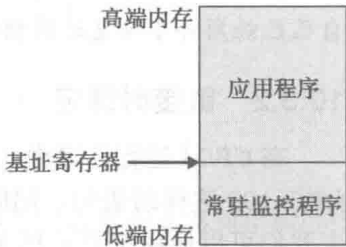


图 10-1 单一进程的操作系统

10.3.4 动态重定位

有关寄存器功能的变化也导致了相应名称的改变，曾经所谓的基址寄存器现在则称为重定位寄存器（relocation register），如图 10-2 所示。加载到该寄存器中的数值不再是一个界限值。相反，可执行程序是按照其好像定位于地址 0 来运行而创建的，并且重定位寄存器中的数值将会加到由对应程序所做的每个内存引用地址上。为此，程序现在没有必要在每次操作系统发生变化时就进行重新链接，因为每个内存引用会在程序运行时自动地重新定位。

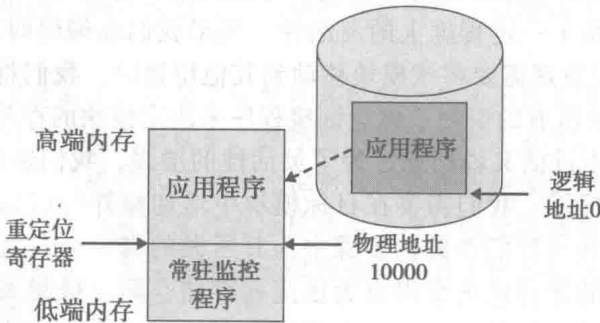


图 10-2 单一进程的操作系统（需要重定位硬件的支持）

213

10.3.5 物理内存空间与逻辑内存空间

这项新的重定位功能引入了一个重要的概念：逻辑地址空间和物理地址空间之间的差异。原来，当我们编译一个程序时，我们是创建了一个程序且该程序将会按照在翻译步骤或链接步骤中为其分配的地址而被加载到内存。可执行程序被编译成是引用了与实际物理存储器（内存）地址一一对应的一系列地址。虽然这一点在硬件演化方面还不太清晰，但实际上存在两个不同的地址空间在被使用。第一个地址空间是在程序执行时由处理器所生成的地址集，这个地址空间称为**逻辑地址空间**。而我们最后将把可执行程序加载到主存中，用来访问这一存储器的地址集则称为**物理地址空间**。当我们引入重定位寄存器时，这一点已变得非常清楚，即程序的逻辑地址空间和它的物理地址空间是不同的，我们从图 10-2 中可以看出。磁盘上的可执行程序是按照生成以零作为低地址的相关地址而被创建的。但是，当程序被定位到存储器中时，其实际上是被加载到物理内存地址 10000 的位置。在程序运行时，存储器地址硬件将会对由程序产生的逻辑地址进行动态的重新定位，具体通过加上重定位寄存器中的数值，从而把逻辑地址空间映射到了物理地址空间中。图 10-3 给出了关于对应进程的一个更为具体的示例。当应用程序在处理器中运行时，其会生成一个内存引用，这可能是程序中下一条指令的地址、一个子程序调用的地址、一个数据项的引用地址或许多其他东西的地址。在这个例子里，相应引用是对应程序的逻辑地址空间中的地址 456。然而，操作系统已经把该程序的起始物理地址（本例中即 10000）加载到了重定位寄存器中。于是，存储器硬件将重定位寄存器中的数值加到该逻辑地址上，从而生成了物理地址 10456。

10.3.6 程序比内存大

随着时间的流逝，内存已经变得非常便宜了。但曾几何时，主存在一个系统的总价格中常常占有非常大的部分。因而，大多数早期的系统往往只有相当小的主存。无论是大型机、小型机还是早期的微机，其主要的内存度量单位为千字（Kiloword）或千字节（Kilobyte，简记作 KB），而不是千兆字节（Gigabyte，简记作 GB），这在当时是相当普遍的情况。正因如此，程序设计人员花费了很多时间试图将更多的功能或更多的信息挤进很小的内存空间里。例如，正是这种压力导致了千年虫（Y2K）问题。当时程序员的自然想法是，既然还有 30 年左右的时间才会是以“19”之外的其他数字打头的年份，为什么要为了每个日期而浪费内存来存储这两个额外的数字呢？今天，我们可能仍然必须要应对拥有有限主存的嵌入式系统，但通常这是出于空间或功率要求的原因，而不是因为存储器的价格。

[214]

10.3.7 覆盖

程序设计人员经常需要为那些在小内存中运行的程序增加某些功能。设计一个由如下三部分组成的程序曾经是（并且依然是）相当普遍的：首先是初始化代码部分，然后是一个主要的计算循环，最后是关于某种总结报告的代码部分。鉴于程序设计人员认识到这类程序的上述部分并不需要同时驻留在内存空间，所以他们决定让有关程序的这些部分在内存中相互重叠或覆盖。图 10-4 就给出了这样的程序。该程序的主体部分只是对实现上述三个组成部分的子程序的一系列调用，为方便起见，不妨设定相关子程序名字分别为

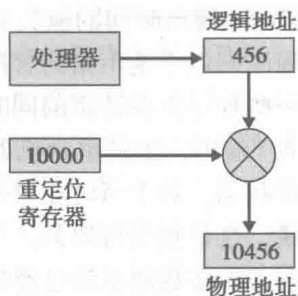


图 10-3 基于存储器地址硬件的逻辑地址空间到物理地址空间的动态映射

“init”、“main-loop”和“wrap-up”。同时，该程序里需要添加上对操作系统加载函数的相应的调用，以便在调用各子程序之前首先实现对应各子程序的加载。程序主体部分常驻在存储器中，而其他部分则是在被调用之前加载到内存的。在精巧复杂的交互式系统中，这可能会有点复杂和棘手。而对于像汇编器或编译器这样的简单程序来说，即便不使用覆盖技术，也可能会比较轻而易举地被加载到内存中，但是，如果将其分解成若干阶段和子程序，则将会允许相关转换机制能够在给定的存储空间中处理较大的源程序。

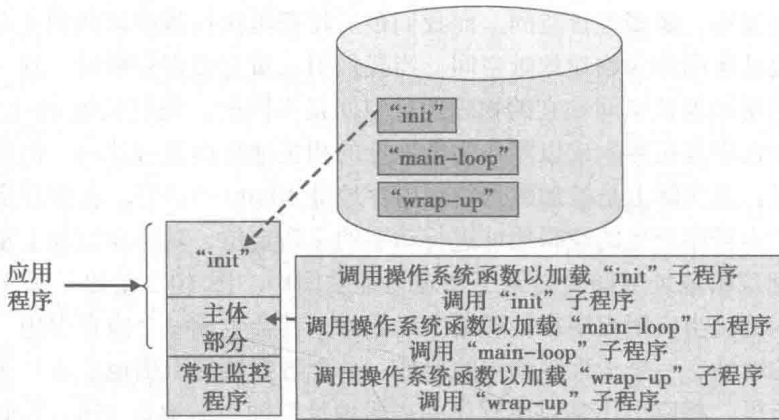


图 10-4 支持覆盖的程序示例

10.3.8 对换

伴随主存价格的下降（相对于计算机的其余部分而言），计算机管理人员着手审视正在发生的一切，并进而意识到，由于只有一个程序运行，所以他们没有让自己非常昂贵的系统得到很好的利用。特别是有些程序会执行较少的输入/输出操作但会计算很长时间，于是外围设备没有得到充分的利用；相反，有些程序则主要完成输入/输出任务而只有非常少的处理器运行任务，故而这些程序在执行过程中常常需要等待输入/输出操作完成，处理器的利用因此受到了极大的影响。在这种背景下，假脱机等一些技术应运而生。具体而言，在单一进程批处理系统中，操作系统可以读取包含有要运行的下一道程序及相关联数据的卡片并将它们存储到磁盘上，且这一读取过程应当与当前作业的执行过程在时间上相重叠。当某道作业试图打印其对应输出时，被打印的各行内容将被存储在磁盘文件中，并且真正的打印过程将与下一道作业的处理过程在时间上相重叠。

然而到后来人们意识到，即便是假脱机技术也是不够的——依然存在大量的处理器周期和输入/输出时间的遗失和浪费。而同时运行多道程序则似乎是个不错的解决方案。也就是说，希望一些程序正在计算的同时而其他程序正在进行输入/输出，于是整台机器将会保持更为忙碌的工作状态。对于当时花费一百万美元购得的机器来说，这是相当可取的。这时候，在计算机的历史上，大多数的系统已经拥有了由磁盘或磁鼓组成的辅助存储器。对换（swapping）技术成为保持若干程序同时运行的第一种技术，相关工作机理如图 10-5 所示。根据图示，程序 A 当前正在

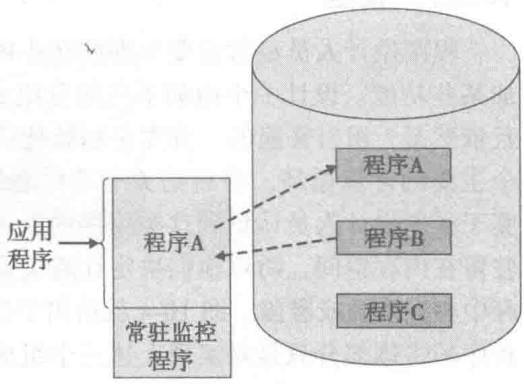


图 10-5 具有对换功能的操作系统示例

主存中运行。假定该程序请求系统要在打印机上打印一行，且这一操作至少需要百分之一秒。在这个时候，我们可以做许多的磁盘输入 / 输出操作和大量的计算，所以我们通过将主存的内容写到磁盘上来把程序 A 换出外存，同时把程序 B 换进内存。接下来我们让程序 B 运行，直到它发出一个针对低速设备的输入 / 输出操作请求，然后我们再将程序 B 交换出内存，并将程序 A 换进内存。对换技术有时也称为滚出 / 滚入 (roll-out/roll-in) 技术。显然，等待输入 / 输出操作所用的时间必须得大于对换所用的时间。然而，利用内存直接存取型硬件，一个连续内存块的对换可以非常迅捷并且对处理器造成的开销很小。

10.4 固定进程数的多进程

然而，即使是对换技术也是不够的，这些价格不菲的机器的所有者们往往想让这些机器完成更多的事情，以便能够对得起他们所投入的资金。为此，操作系统的设计人员开始寻找更好的组织相关处理过程的方法。由于主存继续变得更加低廉，所以有关设计人员开始想办法让多道程序同时保留在主存中，在它们之间交替运行多个程序，而且不用将它们换出内存——毕竟对换是一种需要耗费大量资源的操作。最终，他们认识到重定位寄存器可以支持运行在任何位置的一道程序，而不仅仅是紧靠在常驻操作系统顶部的程序。因此，他们转向考虑如图 10-6a 所示的操作系统内存组织方案。当初，基址寄存器曾经被用来防止应用程序对操作系统的危害。随后，这类寄存器的使用被改成了重定位寄存器，主要用来解决操作系统增长的问题。现在，当操作系统运行多道程序并且一道程序对某个慢速设备执行输入 / 输出操作的情况下，操作系统只需将第二道程序的内存地址放入重定位寄存器中便可开始运行第二道程序。这种情形如图 10-6b 所示。(其实，操作系统所做的事情远远不止这些，但在这里我们只是关注内存方面的一些管理操作要点。)

216

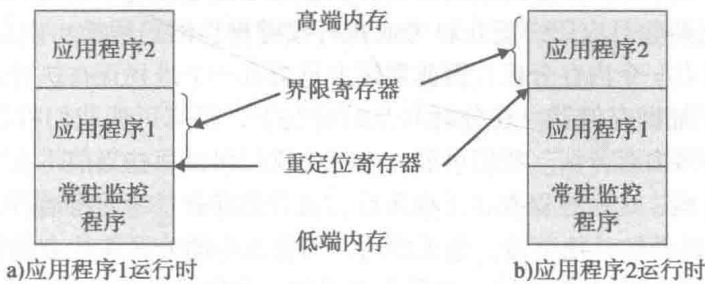


图 10-6 固定进程数的多进程操作系统示例

此刻我们已经进展到：还有其他的应用程序也在主存中运行，所以有必要针对由此引发的新问题进行研究，以使相关应用程序不会彼此危害到对方。相应的解决方案就是增加一个界限寄存器 (limit register)，用来建立一个上限 (upper bound)，如果超出这一上限，对应用程序将无法寻址，就像其不能对重定位寄存器所设定的程序起始地址以下的地址进行访问一样。有人可能会认为这只不过是包含程序最高地址的另一个寄存器而已，但是由于我们后面将要说明的有关原因，实际上这个寄存器里面几乎普遍性地包含的是对应程序的大小，而不是程序的最高地址。有关硬件将会即时地把这一界限值加上重定位地址值，从而建立起当前程序寻址的上限。与下限 (lower bound) 的情形一样，如果程序试图访问超出由界限寄存器所设置的上限范围的内存，那么有关硬件将会产生寻址错误中断，进而操作系统将会中止对应用程序。因此，当操作系统从运行这道程序转向运行另一道程序时，其必须同时对重定位寄

存器和界限寄存器进行设置。

10.4.1 内部碎片

当安装这种类型的操作系统时，管理员应当决定为每个程序区域或内存分区（partition）留出多大的内存。同时，在系统运行的过程中，操作系统不应更改这些内存分区的大小。对于较早以前的操作系统模型来说，一道程序可能不会使用所有的内存。而如果有关程序没有用完所有内存，我们也不会对此有所牵挂。现在，操作系统正在试图把更多的程序放进同一个内存空间里。如果我们为每个内存分区留出 100KB，而我们想要运行一个只需要 6KB 的程序，那么我们势必会浪费掉对应内存分区中的其余空间。这部分未被使用的空间称为内部碎片（internal fragmentation），如图 10-7 所示。我们可能会设置一个或两个小的内存分区用来运行一些短小精干的作业，还会设置一个或两个较大的内存分区用来运行我们的大型的应用程序。这样将有利于最小化由于内部碎片而浪费掉的内存空间。如果管理员可以更加精巧地设置有关内存分区的大小，那么正在运行的程序应当接近于填满主存空间，并且我们将会会有更好的机会来保持昂贵的硬件得到充分的利用。

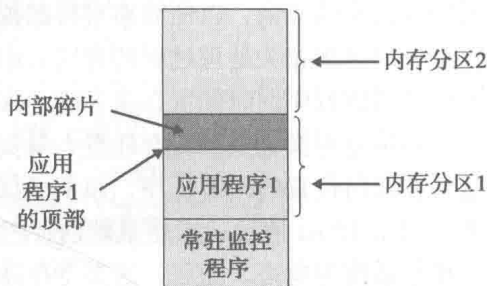


图 10-7 内部碎片

10.4.2 分时共享

利用对换技术的另一种情况是在被设计为以所谓分时共享（time sharing，或简称分时）方式来支持许多用户通过各自终端使用的系统中。在有关用户以交互方式编辑程序并测试相应程序的过程中，其中绝大多数的时间里，有关进程是在等待对应终端上的用户。在这种情况下，系统可以把那些对应用户正在思考或执行按键操作的进程换出内存。在第 10.3 节所描述的例子中，只有一个内存分区，因此实际上只会会有一个进程正在执行，任何其他的进程都可能被换出到了辅助存储器。在分时共享的情况下，很有可能我们将会拥有若干内存分区，或许甚至是许多内存分区。我们可能在内存中仅仅保留那些当前不在等待对应用户完成一行输入操作的进程，这些进程或者正在运行，或者已准备就绪即将运行，或者正在等待除了终端输入/输出以外的其他事情。毫无疑问，固定大小的分区往往会浪费内存空间。请回想一下我们刚刚讨论过的内部碎片，在那个例子里，我们只有唯一的一个分区的内部碎片。而现在呢，我们在每个分区里都有内部碎片。我们希望能够利用那些碎片。如果我们节省下了足够多的内存，那么也许我们就能够运行另一个程序从而保持处理器处于更为忙碌的状态。虽然这些技术在当时已经足够好了，但是对于现代的分时系统来说，则通常会使用下一章中所描述的技术。

10.5 可变进程数的多进程

一种部分地解决了这种内部碎片问题的方案是使内存分区的大小或数量不再固定不变。相应地，运行一道程序需要多少内存，我们使用多少内存。因此，我们要求程序设计人员在运行程序之前能够估计出对应程序所需主存的最大数量。这样，当我们启动一道程序时，我们就可以为对应程序分配刚好那么多的内存。如果该程序试图使用比程序员所声称的其所需主存最大数量更多的内存，那么操作系统将会按发生了错误而中止相应程序。在一道程序

结束时，操作系统将会再次把相应内存区域置为空闲可用状态（或“未使用的”状态），从而使其可用来运行另一道程序。这部分内存空间通常被称为空穴（hole），有时或称为外部片段（external fragment）——我们当前根本不使用的内存块。在图 10-8a 中，我们可以看到系统当前正在运行四道应用程序的情形。在图 10-8b 中，我们看到应用程序 2 和应用程序 4 均已经结束，因此它们当初运行时所在的空穴现在又可以用来运行其他的程序。操作系统通常需要保存一张可用的空穴列表。在图 10-8c 中，我们看到操作系统已经启动了应用程序 5，而其占用了当初应用程序 2 运行时所在的空穴的一部分空间，故而现在形成了一个更小的空穴。

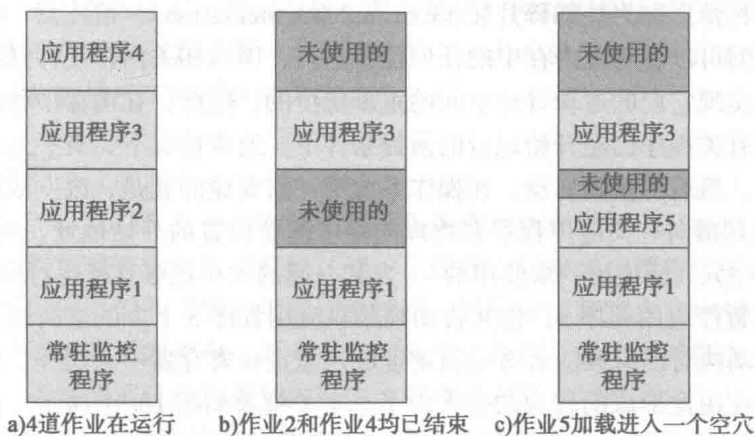


图 10-8 可变进程数的多进程操作系统示例

现在假设操作系统有另外一个程序要运行，而存在很多空穴可供选择。那么，操作系统应该选择哪个空穴呢？与我们在本书中研究的大多数算法类一样，第一种算法就是直接扫描整个空穴列表，并选用其间我们所发现的第一个足够大的空穴来运行相应程序。这种算法称为**首次适应**（first fit）算法，其具有简单易懂且易实现的优点，不过这种算法可能并不是最好的选择。另一种算法则是选用可以容纳我们想要运行的程序的最小的空穴，该算法称为**最佳适应**（best fit）算法。从直觉上来说，最佳适应算法非常具有吸引力——那样我们就可以保证浪费的主存最小呀！然而令人遗憾的是，这种算法要求我们或者扫描整个空穴列表，或者按空穴大小次序来维护该空穴列表，无论哪一种方式都需要额外的处理开销。

如果我们一般的程序需要 10MB，而我们有一个程序需要 8MB 来启动运行，同时我们现在拥有一个 12MB 的空穴和一个 18MB 的空穴，那么我们应该怎么办呢？如果我们使用 12MB 的空穴，那么我们将会形成和留下一个 4MB 的空穴，一般来说这样大小的空穴我们是无法再使用的。而如果我们使用 18MB 空穴的一部分，那么我们将会形成和留下一个 10MB 的空穴，一般来说这样大小的空穴我们是能够使用的。因此，接下来的这种算法认为，我们应当使用最坏适应的那个空穴，理由在于该算法形成和留下了最大（因而是最有用的）的空穴，称之为**最坏适应**（worst fit）算法。同样，这种算法要求我们或者扫描整个空穴列表，或者按空穴大小次序来维护该空穴列表。

219

建立在首次适应算法基础上并有一些轻微改动的算法称为**循环首次适应**（next fit）算法。对于这种算法来说，我们每次搜索并不是从空穴列表的前面开始，并且我们不用保持空穴列表的有序性。准确地说，我们总是从上次搜索停下的地方开始下一轮的搜索。首次适应

算法往往倾向于把空穴列表前面的那些空穴分解地七零八落，所以我们常常会形成一群很小的空穴，这些空穴需要我们不断地浏览审查但很少能够使用。循环首次适应算法则倾向于把这种碎片化分散到整个空穴列表的各个地方。在实践中，最坏适应算法被证实是效果最差的，最佳适应算法或者循环首次适应算法要更好些，并且循环首次适应算法非常容易实现。

现在假设我们拥有两个分别为 5MB 的空穴，并且我们有一个声称其可能需要 8MB 来运行的进程。不难看出，我们在两个空穴中共有 10MB 的空闲内存块，二者合到一起有足够的空闲内存支持相应进程的运行。但是，有关的空闲内存并不在一块里面，所以我们无法运行相应程序，这种情形称为**外部碎片化**（external fragmentation）。请注意，我们的进程是可重定位的——它们可以在物理内存中的任何位置运行，因为相关内存硬件是在有关进程运行的时候，动态地实现它们的逻辑寻址空间的重新定位的。因此，在内存中移动一道程序是可以的，即使是在有关程序已经开始运行的前提条件下。通常情况下，有关进程应被挂起，移动到另一个位置，然后再重新启动。而操作系统唯一需要做的就是，改变放置在重定位寄存器中的数值，使其指向有关应用程序在物理内存中的新位置的开始地址。例如，在图 10-9a 中，如果有两个空穴（标记为“未使用的”）合到一起的大小足够支持运行应用程序 6，那么操作系统就可以暂停应用程序 3，将其移动到紧靠应用程序 5 上方的空间区域，并且每当应用程序 3 再次启动执行的时候应将对新地址放入重定位寄存器中。这样，系统就可以启动应用程序 6 运行在由此形成的较大的空穴中了。有关结果如图 10-9b 所示。这一过程称为**紧凑**（compaction）。自然地，相关情况往往比这个简单的例子要复杂得多，并且经常需要几个程序同时迁移和重新定位以便能够形成和找到足够大的空穴来运行我们想要运行的那道程序。有人可能很欣赏，当操作系统在存储器中移动相关程序时，期间没有任何操作或工作是代表那些应用程序而完成的。在此，为了可以让更多的作业能够并行运行，操作系统选择和付出了处理器和内存带宽。

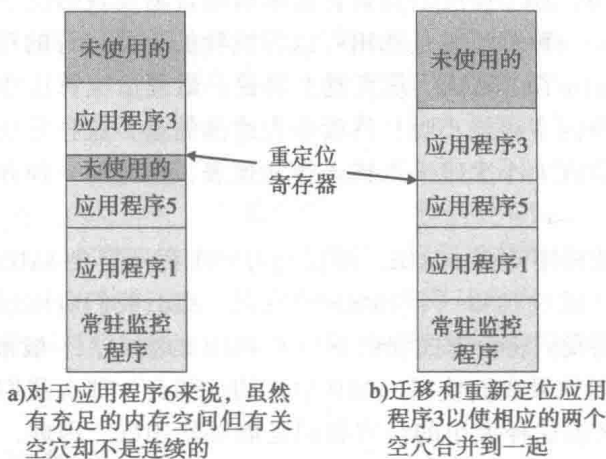


图 10-9 紧凑

现在，我们就可以领会为什么重定位硬件是用长度而不是用程序的上界地址作为上限的理由了。如果有关硬件使用了上界地址，那么当我们重新定位一道程序时，我们还必须得重新计算上界值。这并不是一项特别复杂的计算，并且也不需要经常这样做。然而，如果有关硬件能够以其他方式很好地工作，那么我们很庆幸可以不必这么做。

关于紧凑处理的一种复杂情况可能是，输入/输出硬件或许没有使用有关重定位硬件。换句话说，输入/输出操作使用的是物理地址而不是逻辑地址。这意味着，如果一个进程有输入/输出操作正在进行且尚未完成，那么我们就不能在内存中对其进行移动。因此，启动了输入/输出操作的进程可能必须临时标记为不可移动的。

我们还可能依然会遭受到内部碎片的影响，而这种情况的发生则源自于我们的空穴可能会变得越来越小的现实。一般来说，让操作系统来监测好多非常小的内存块经常会影响到系统的效率。因此，应当设定关于操作系统试图管理的某种最小内存量值，通常这一最小量值是在 256B 到 4KB 之间的范围内。当有关程序启动时，该程序将被分配给一个大小为相应最小内存量值（译者注：不妨称大小为相应最小内存量值的内存为单位块）的整数倍的内存块。平均而言，每个程序将不需要其最后一个单位块的一半大小的空间，所以这部分内存将会被浪费掉——可变进程数情况下所产生的内部碎片。

10.5.1 动态加载

使用覆盖技术时，我们没有一次性地将整个程序都加载到主存中。相反，程序设计人员明确指定，什么时候加载有关的覆盖代码区域，以及什么时候调用覆盖代码区域中的相应例程。系统应当也可以做类似的事情。当操作系统把一道程序加载到主存中时，它也可以只是把程序的主体部分加载到了内存中。而为了访问各个子程序，操作系统可能会设置和利用一张表，其中标明了哪些例程已经加载到内存中，哪些例程尚未加载到内存。许多程序大致遵循“80-20”法则——一道程序中 80% 的代码是为了应对那些只有 20% 的时间才会发生的情况。因此，如果我们在程序第一次启动时不加载有关的子程序，那么我们可能永远也不需要加载它们。这样，程序就会启动得更快些。而如果程序稍后会调用有关例程，那么我们可以在那时候再加载对应例程（译者注：称之为动态加载），并且我们将会为这种延缓付出很小的代价——一小部分的内存用于保存有关表格，同时每当我们第一次调用有关例程时会执行一些额外的指令。

10.5.2 动态链接库

然而，我们甚至可以把绑定操作也向后推迟一步。也就是说，我们甚至可以推迟有关子例程与主模块之间的链接。这样库例程（library routine）本身就不再成为可执行程序的组成部分。相反，我们需要原样保留编译器所生成的针对库例程的符号引用（symbolic reference）。于是，就像动态加载一样，如果有关例程从未真正引用过，那么我们不仅不会将其加载到内存中，而且我们甚至没有将对应符号绑定到一个逻辑地址上。这里，我们原样保留的这些子例程是在通常被称为动态链接库（Dynamic Link Library, DLL）的特殊的库中。在 Linux 和大多数其他的 UNIX 变体操作系统中，这样的库被称为共享对象库或动态库，并且通常以“.so”作为其名称的一部分。当引用此类库中的子例程时，操作系统将会在程序执行时才会把对应例程加载到存储器中并绑定相应链接。

注意，我们还同时从这种机制获得了一些其他的好处：

- 由于有关子例程并非可执行程序的组成部分，所以相应程序比较小，故而就只会占用磁盘驱动器上的更少的空间，并可以更快地加载到内存中。
- 通常情况下，我们将会拥有使用相同的库模块的许多程序。其中的某些库模块使用的频率非常高，不夸张地说，它们将会被硬盘驱动器上的成千上万的程序所引用。

然而此时，我们却仅仅需要拥有一份代码副本，由此便可以为我们节省大量的磁盘空间。

- 如果有关库中的某个模块的错误被修复，那么只需要修复和整理相应的一个库例程，并将其加载到系统上。这将会实现所有引用相应动态链接库的每个程序中的对应错误的自动修复。

对于应用程序软件的开发人员来说，上面的最后一项特征可以算得上是一份巨大的恩赐，因为这意味着，如果修复工作是由操作系统制造商针对系统库（system library）来实施，那么应用程序开发人员就不必使用新库来进行应用程序的重新组装和生成，也无须将可执行程序重新分发给正在运行对应平台的每个客户。而如果有客户打进电话来，所抱怨的是与另一家供应商所提供的一个动态链接库有关系，应用程序开发人员也只是需要解释一下，相应问题是出在系统库中，并且该库模块在版本 x.y.z.1.5 中就已经进行了修复，该库模块可以从该库的供应商的网站下载，……。另外，如果应用程序的供应商真的特别幸运，有可能客户是先其他的某个应用程序那里发现的此类问题，并在该客户遭遇到他们的应用程序的问题之前就已经下载了相关已完成修复处理的新库。

遗憾的是，动态链接库存在一个问题。具体来说，当某软件包的开发人员使用一个动态链接库中的一组特定功能时，他们的代码也可能依赖对应库的特定版本中的错误修复。他们应当希望确保他们正在使用的这组功能和错误修复是包含在安装了该软件包的任何系统上的可用的库版本中。因此，软件包安装可以包括一个库版本至少是与该软件包的供应商开发时所用的库版本一样新。遗憾的是，目标系统可能已经包含了由另一个软件包所安装的更高版本的库，该软件包依赖于该更高版本的库中的功能或错误修复。安装较旧的版本将会导致那个已安装的软件包出现故障。为此，突然出现故障而停止工作的软件包的供应商可能会十分惊讶地收到由此导致的支持服务请求，而当问题最后终于得到解决但时间已经被白白浪费在解决一个与他们公司所做的任何事情没有哪怕一丁点儿关系的问题上的时候，他们势必会恼羞成怒，显然这是可以理解的。当然，安装软件应该检查确认正在安装的任何动态链接库要比已经安装的对库的版本更高。遗憾的是，这项检查并不总是可以做到或者有可能做错了，这类问题俗称为**动态链接库地狱**（DLL Hell）。较新的操作系统版本允许应用程序为某个动态链接库指定版本号，因此这类问题就通过允许一个系统携带单个库的多个版本而降低到了最低限度。这样一来，必然会占用一些额外的空间，但与让有关库作为每个使用该库的应用程序的一部分所消耗的空间比较起来，可就差远了。

222

10.6 小结

在本章中，我们讨论了关于主存可由操作系统进行管理的许多方面的内容。一开始，我们讨论了为什么操作系统管理内存，并阐明其目的是为了支持所需空间大于机器主存空间大小的有关程序的运行，并允许尽可能多的程序一起运行。然后，我们讨论了软件开发周期，以及关于地址绑定的各种可能的时机。接下来，我们渐进式地探讨了更为复杂的内存模型，从单个进程开始，一直到涵盖固定的和可变的多进程处理的连续内存组织方式。有关讨论期间，我们还重点说明了这些操作系统技术所需的硬件支持。最后，我们还用一节内容介绍了有关例程的动态加载的优点和不足。

在下一章中，我们将会讨论通过分页和分段方式来解决内存管理相关问题的一些现代的方法。

参考文献

- Daley, R. C., and J. B. Dennis, "Virtual Memory, Processes and Sharing in Multics," *CACM*, Vol. 11, No. 5, May 1968, pp. 306–312.
- Dennis, J. B., "Segmentation and the Design of Multiprogrammed Computer Systems," *Journal of the ACM*, Vol. 12, No. 4, October 1965, pp. 589–602.
- Kilburn, T., D. J. Howarth, R. B. Payne, and F. H. Sumner, "The Manchester University Atlas Operating System, Part I: Internal Organization," *Computer Journal*, Vol. 4, No. 3, October 1961, pp. 222–225.
- Knuth, D. E., *The Art of Computer Programming: Fundamental Algorithms*, Vol. 1, 2nd ed. Reading, MA: Addison-Wesley, 1973.
- Organick, E. I., *The Multics System: An Examination of Its Structure*. Cambridge, MA: MIT Press, 1972.

本章的参考文献与下一章的参考文献有着相当多的重复。

习题

- 10.1 操作系统必须致力于管理主存的根本原因是什么？
- 10.2 从程序创建开始到其在内存中执行所需经历的 5 个步骤是什么？
- 10.3 术语“绑定”是什么意思？
- 10.4 在习题 10.2 中所列出的 5 个步骤之中，可以实施绑定的步骤有哪些？
- 10.5 逻辑寻址空间和物理寻址空间有什么区别？
- 10.6 为了试图同时运行多道作业，我们创建了一些固定分区。在此我们遇到了内部碎片的问题。请描述这个问题。
- 10.7 固定分区的一种替代方案是允许可变分区。但其在最小化内部碎片的同时，却又带来了一个新的问题，即外部碎片的问题。请描述这个问题。
- 10.8 关于外部碎片，我们采取了什么措施？
- 10.9 当采用可变分区进行内存管理时，我们可能拥有若干空穴均足够大到可以运行我们想要运行的下一道作业。我们列出了 4 种算法，用来从足够大到支持对应进程运行的那些空穴中选择要使用的空穴。请给出这些算法的名字并加以简明扼要的说明。
- 10.10 阐述动态加载和动态链接之间的区别。
- 10.11 动态链接拥有一个巨大的优势和一些较小的优势。请阐明这个巨大的优势和两三个较小的优势。

高级的内存管理

这一章将继续讨论内存管理技术。特别地，本章涵盖了现代系统中所使用的更先进的技术，即高级的内存管理技术。第 11.1 节讨论了上一章相关机制所引发的有关问题以及为什么要开发新技术。

第 11.2 节描述了有关分页硬件的功能以及其如何实现逻辑寻址空间和物理寻址空间的进一步隔离。第 11.3 节则讨论了一种称为分段的备选硬件机制。接下来，第 11.4 节介绍了如何把分页和分段结合到一起来进行使用。在第 11.5 节中，我们转向讨论请求分页的主题——只有当相应页面被访问时才将它们装入内存，以及伴随这种技术而出现的一些问题。第 11.6 节介绍了几种特殊的高级内存技术，而第 11.7 节则对本章进行了总结。

11.1 为什么需要硬件的辅助支持

225

采用连续内存分配的多进程处理往往会导致外部碎片，于是有些时候，即使是有足够的内存可用来运行有关程序，我们也无法运行它们，这种情况下便会造成内存和处理器资源的浪费。在上一章中，我们看到，尽管我们可以设法缓解这个问题，但是相应解决方案需要运行紧凑例程，而相关处理过程则是对处理器和内存的非生产性的使用。为了消除这个问题，我们需要进一步对程序可见的内存地址空间（逻辑地址）和硬件所使用的地址空间（物理地址）进行隔离，从而使一道程序的各个组成部分不一定非得装在一片连续的内存空间中。要实施这项隔离需要硬件的辅助支持。有关问题可以采用若干种不同的方法，而这些方法将在以下各节中分别展开介绍。

11.2 分页

早些时候，我们讨论了把逻辑寻址空间与物理寻址空间区别开来的想法。我们改造了有关内存管理部件（Memory Management Unit, MMU）以使其支持这种想法。进而，我们不再使用基址寄存器来进行地址的检查，而是使用它来重定位地址。这便允许我们把任何程序放置在内存的任何地方——动态重定位（dynamic relocation）。然而我们发现，允许可变大小的程序在内存中进进出出将会导致我们拥有内存的外部碎片，并需要花费宝贵的处理器时间来实施紧凑处理。遗憾的是，从紧凑并非用户试图要做的任何事情这个意义上讲，它还算不上什么“有用的工作”。它仅仅是操作系统谋划使相关事物在整体意义上更好地运作的一项任务而已。于是，最后终于又开发出了另外一种解决方案——我们把内存划分成固定大小的物理块，从而不再把一个应用程序所需的在一个大的分段中的整个空间都分配给它，而是分配足够多的较小的物理块给程序以满足它的需要。鉴于我们要求有关内存管理部件分别独立地针对每个物理块进行动态的重定位，所以我们分配的这些物理块并不需要是连续的——它们可以在内存中的任何地方。这种技术被称为分页（paging）。同时，这也意味着我们不得不让我们的内存管理部件变得更加复杂。

我们将会把我们的物理地址空间划分为统一大小的物理块，称之为页框（frame，或称

为物理块，有时也称为内存页面，甚至简单地称为页面)。与此同时，我们从概念上将逻辑寻址空间划分成一系列同样是统一大小的称为页面 (page, 或称为分页) 的块，且页面的大小与页框相同。通常，这些块的大小是 512B 到 8KB。对于按字节寻址的机器来说，一个块中的字节数一般总是 2 的整次幂。今天，一种常见的页面大小为 4KB，不过，随着内存和硬盘驱动器的空间的增大，将意味着未来我们很可能会看到更大的页面大小。图 11-1 就每个地址引用的重定位过程进行了展示。

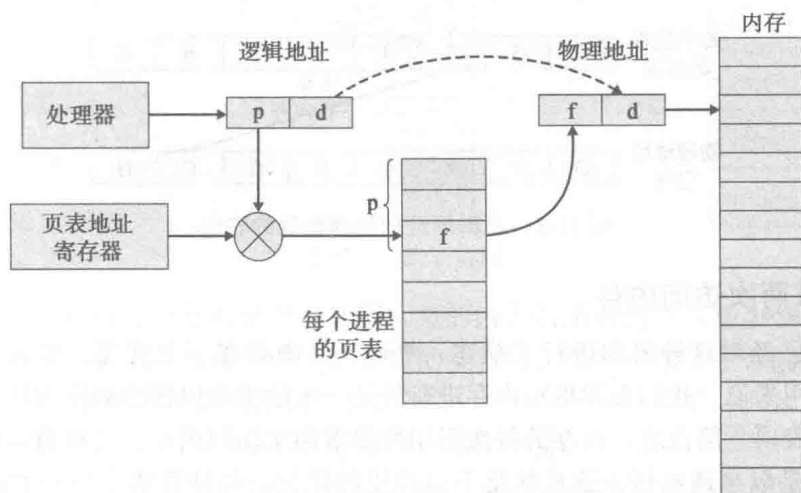


图 11-1 基于分页的内存访问

在图中，我们看到处理器生成一个内存地址。一般来说，程序忽略了内存以单独的页面来进行管理的事实，因此，这些地址仅仅被认为是逻辑地址空间范围内的一个二进制数。有关地址可能是（按次序而言的）下一条指令的地址、要跳转执行的子程序的地址或者是对数据项或堆栈的引用的地址。在这里，有关地址引用的目的并不重要。就像前面一样，我们把这一地址称为逻辑地址。然而，内存管理部件将该地址看作是由两部分组成的，这里分别标记为页号 (page number) p 和偏移地址 (displacement) d 。此偏移地址是页框内特定字节的地址。如果我们的页框的大小是 4KB，那么我们的偏移地址字段长度就应是寻址一个页框内的每个字节所需的合适的大小，也就是 12 个二进制位。当我们将逻辑地址重新定位到一个物理地址时，我们在对应页框中所寻址的偏移地址与我们在相应页面中所寻址的偏移地址是相同的。为此，我们无须改变逻辑地址的偏移地址部分。逻辑地址的剩余部分是页号。我们需要重定位的就是页面，因此我们应当查找关于重定位地址的一个页表 (page table) 以确定相应的页框。我们将会拥有一个寄存器用来保存当前正在运行的进程的页表的内存地址，该寄存器称为页表地址寄存器 (page table address register)。有关内存控制部件将会根据在处理器中运行的进程所产生的逻辑地址中的页号以及页表地址寄存器中的数值执行加法操作，从而查找和定位到相应的页表项上。存储在页表的该位置的数值就是我们正在试图访问的特定页框的重定位地址。在这个例子里，它显示为数值 f 。数值 f 应当与我们已经拥有的用来寻址物理内存中特定字节的偏移地址组合到一起，从而生成相应的完整的物理地址。

图 11-2 显示了一张更为完整的页表。我们忽略了地址的偏移地址部分，而只考虑页面是如何映射到页框的。我们看到，当前正在运行的进程的逻辑地址空间被划分成了分别标记为 A~H 的若干页面，例如，第三个页面被标记为 C。如果处理器生成了一个针对此页的引

用，那么内存管理部件将通过查看该进程的页表中的第三项来进行这一地址的转换。这里，我们看到该项中包含的页框号为7。因此，内存管理部件将具体查看页框7来寻找我们正在访问的信息。当然，在实际的系统中，一个进程的页框将被分散开来并与来自其他进程的页框混杂到一起。

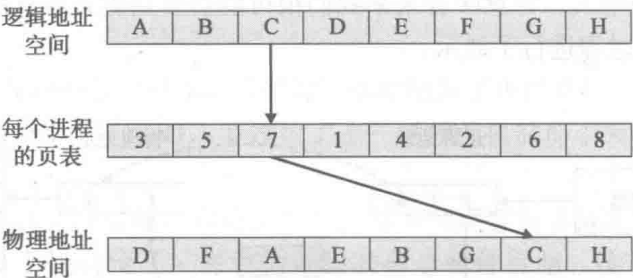


图 11-2 逻辑地址到物理地址的映射

227

11.2.1 要求两次访问内存

上面我们已经对这种机制进行了描述，然而，其中存在一个问题。也就是说，就针对内存的每次引用来说，我们必须得对内存进行另外一次的引用以便找到作为该页面的重定位因子的相应页表项。简言之，内存的每次引用均要求两次访问内存。这样将会使我们的进程在访问内存时是以半速运行，这显然是不可接受的代价。与计算机系统许多其他事情一样，我们的解决方案是让内存管理部件缓存最近的几个重定位因子，从而使我们不必非得得到内存中查找它们就可以再次使用它们。这种缓存是通过称为**快表**（translation lookaside buffer, TLB，又称为转换后援缓冲器）的一种特殊类型的硬件设备来完成的。快表是一种叫过若干个名称的电路，有时被称为内容可寻址存储器（Content Addressable Memory, CAM）或关联存储器（associative memory，又称为联想存储器）。这个电路元件的精华是，当它试图检查确认自己是否包含某个页号时，其所有表项的搜索是并行展开的。这意味着，快表中的表项不必非得某种顺序来保存，因为它们都是在同一时间进行比较的。如果我们试图访问的页面最近曾被访问过，那么它应当包含在快表中，故而相关查找结果将会被很快返回——也许比我们访问主存中的页表要快上 100 倍。显然，快表是一个复杂的电路。因此，它们的容量通常相当小。在目前的机器上，它们一般很少会超过 1000 个表项，而且常常会更少。然而在通常情况下，这足以使大多数的进程在大部分的时间里都可以在该缓存中找到所需的信息。快表的使用机制如图 11-3 所示。

11.2.2 有效的内存访问时间

有一个公式，我们可以用来评估有关快表对计算机执行速度的影响。根据该公式，我们将会计算**有效访问时间**（Effective Access Time, EAT）。该公式中用到了快表查找的速度（我们记为 E）、主存引用的速度（我们记为 M）以及关于我们在快表中能够找到我们所引用的页号的情形在所有时间内所占的百分比（我们记为 A）。这个百分比通常被称为**命中率**（hit ratio）。不言而喻，我们在快表中找不到所引用页面的百分比（即快表的未命中率）应当是 1-A。举例来说，如果我们在所有时间里可以命中 80%，那么我们在所有时间里将会错失 20%，即未命中率为 20%。当我们在快表中找到对应页号的情况下，那么内存引用过程耗费的时间将会是 E+M，其中搜索快表的时间为 E，而用来进行正常的内存引用的时间为 M。当

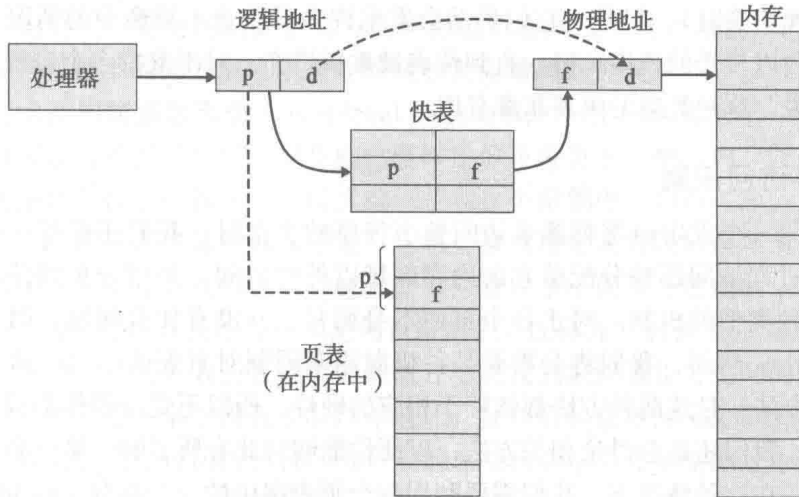


图 11-3 快表机制

我们在快表中找不到对应页号的情况下，那么总的内存引用时间应当是 $2 * M$ ，即两次内存引用的时间，其中一次用于从页表中获取相应的页框号，另一次用于正常的内存引用。为此，综合而言，有效的内存访问时间应当是：

$$EAT = A(E + M) + (1 - A)(2 * M)$$

例如，设定我们的快表查找时间 (E) 为 $5ns$ ，而我们的内存访问时间 (M) 为 $100ns$ ，且我们的命中率 (A) 为 80% 。那么，有效的内存访问时间就应当是 $0.8(100 + 5) + (1 - 0.8) * (2 * 100)$ ，也就是 $124ns$ 。同正常的内存引用相比，这里有约 25% 的降速。

根据有关硬件设计，快表查找可能是发生在第一次内存引用被触发的同时。如果快表查找不成功，那么主存的引用将会继续。在这种情况下，刚才给定的公式才是适用的。但是，某些其他的硬件可能是直到快表查找失败时才启动第一次内存引用。在这种情况下，有效的内存访问时间公式应当是：

$$EAT = A(E + M) + (1 - A)(E + 2 * M)$$

一般来说，我们的快表越大，其命中率就越高。例如，使用与上面的例子相同的基础数据但命中率修正为 90% ，那么有效的内存访问时间应当是 $0.9(100 + 5) + (1 - 0.9) * (2 * 100)$ ，也就是 $114.5ns$ 。同正常的内存引用相比，这里仅仅只有不到 15% 的降速。遗憾的是，这是一个硬件设计决策，而不是一个软件问题或者甚至系统的购买者可以做出的决定。还有，快表与内存不同。具体而言，快表一般不是可升级的，快表是内存管理部件本身不可分割的一部分并且通常嵌入在处理器芯片中。

请注意，每个进程拥有相同的逻辑寻址空间——相应地址从 0 开始，并向上递增，一直到地址取值为程序大小。在大多数系统上，快表硬件并不关心是哪个进程正在运行。当一个进程运行时，快表将填充与当前运行进程的页号相对应的页框号。当我们切换到另一个进程时，操作系统必须告知硬件忘掉所有当前的页框号，因为新的进程将会拥有不同的页框号需要映射到前一个进程曾经使用过的相同页号上。因此，在我们执行上下文切换到新的进程后，对于一开始发生的几个内存引用，我们往往不会在快表里命中，所以我们的进程将会以半速来执行有关的内存引用指令。这也是为什么我们不想频繁地切换进程以及为什么切换线程比切换进程更快的一个原因所在。一些硬件设计确实提供了地址空间标识符 (Address Space Identifier, ASID)，将其与相应的页框号一起存放在缓存中，而且相关设计不再要求

刷新快表（进程切换时）。但是，它们仍然会发生许多的快表未被命中的情况，故而会在一个较短的时间内以较慢的速度运行，直到快表被重新填充。对于支持多个进程以并行方式运行的处理器来说，这种类型的快表非常有用。

11.2.3 内存访问控制

当我们使用一个重定位寄存器来访问整个程序的主存时，我们还拥有一个界限寄存器用来禁止一个进程访问那些分配给它的内存区域以外的空间。采用分页硬件的情况下，我们也将需要一种类似的机制。对于各个页面本身而言，并没有什么问题，因为它们通常都具有固定的大小。然而，我们将会需要某种机制用来限制对页表的访问。这一问题基本上存在两种解决方法，且这两种方法都依赖于相应的硬件，所以不是由操作系统的设计人员所决定的，不过，我们还是会讨论相关方法，以便你能够对此有所了解。第一种方法是使用固定的页表大小。在这种情况下，我们需要利用每个页表字中的一个有效（valid）位来指示对应页表地址是否有效。例如，如果我们拥有一个固定大小为 10 个表项的页表并且该进程在逻辑地址空间中仅占用了三个页面，那么我们将用对应的内存页面编号（即页框号）地址来填充前三个页表项，同时把这三个页表项的有效位设置为“有效”（譬如具体表示为二进制位“1”）。而对于该页表中的其余表项，由于它们并没有包含对有效页面的引用，所以我们将其相应的有效位设置为“无效”（譬如具体表示为二进制位“0”）。当内存管理部件访问页表中的任何表项时，如果对应表项的有效位是设置为“无效”，那么它将会产生一个内存寻址错误。

内存地址控制的另一种方法是使用具有可变大小的页表。在这种情况下，我们将会拥有一个页表长度寄存器（page table length register）。对于单个的重定位寄存器而言，我们所拥有的长度寄存器用来说明主存中的进程的长度。而页表长度寄存器的作用则像它字面上的意思一样，具体用来保存一个进程的最大有效页号。如果处理器在进程运行时所生成的地址包含了一个比页表长度寄存器中的数值还要大的页号，那么有关硬件将产生一个寻址错误中断，因为该进程生成了一个针对该进程逻辑地址空间以外的页面的引用。现在，大多数系统都采用有效位方案，相关理由我们将会在后面看到。

页面访问保护

除了限制内存寻址之外，分页机制还允许操作系统限制针对各种页面可以采取的访问方式。例如，有关硬件可以被设置为仅允许对一个页面进行读取访问，或者仅允许执行访问。为了有效地利用这一机制，编译器（和汇编器）必须能够强制对应的链接器按页面范围来放置可执行文件的各个部分。这样，对应模块的数据部分就可以被标记为允许读写但不可执行。类似地，程序代码可以被标记为仅允许执行。不过，这种机制还存在一些问题需要解决。例如，就堆栈而言，似乎看起来不应当赋予堆栈上的项以执行访问方式。但是对于 Java 虚拟机来说，把 Java 程序字节码编译为堆栈中的指令并在那里执行它们则是经常会有事情。

11.2.4 大页表

在具有现代操作系统和现代编译器的现代机器中，程序正变得非常庞大。这意味着页表也非常大。此外，事实证明，在许多情况下页表是稀疏的，故而意味着位于逻辑地址空间范围内的有关页表项大部分并没有指向一个有效的页框。稍后，我们将讨论会发生这种情况的

一些原因。无论如何，就分配给页表自身的内存进行管理这一任务来说，已变得越来越难。不过，还是有若干种不同的方法可用来处理这些大而稀疏的页表。

第一种技术是构建多级页表（multilevel page table）。图 11-4 显示了一个两级页表——其本质就是对页表又进行了分页。与我们已经讨论的单级页表一样，内存管理部件把由处理器产生的逻辑地址看作是由若干部分组成的——在这个示例中，即由三部分组成。和前面一样，我们拥有对应的页面内部的偏移地址，它将会被直接带入后续转换环节并用作相应的页框内部的偏移地址。现在，我们进一步将页面编号看作是由两部分组成的，图中具体标记为 p1 和 p2。其中，p1 将被有关硬件用来访问顶级页表（也称为一级页表），就像前面所描述的那样。然而，存储在顶级页表的表项中的数值并不是我们最终真正要访问的内存地址所对应的页框号，而是对应于一个二级页表的内存地址。页号的剩余各位（这里标记为 p2）将用来在所选择的二级页表内进行访问，且二级页表的表项将会给出在原始地址中由 p1 和 p2 所表示的页号相对应的页框号。该页框号将与原始的偏移地址一起使用，用来访问物理存储器中的相应请求的内存位置。数字设备公司（Digital Equipment Corporation，DEC）的 VAX 系统就曾使用了两级分页体系结构。

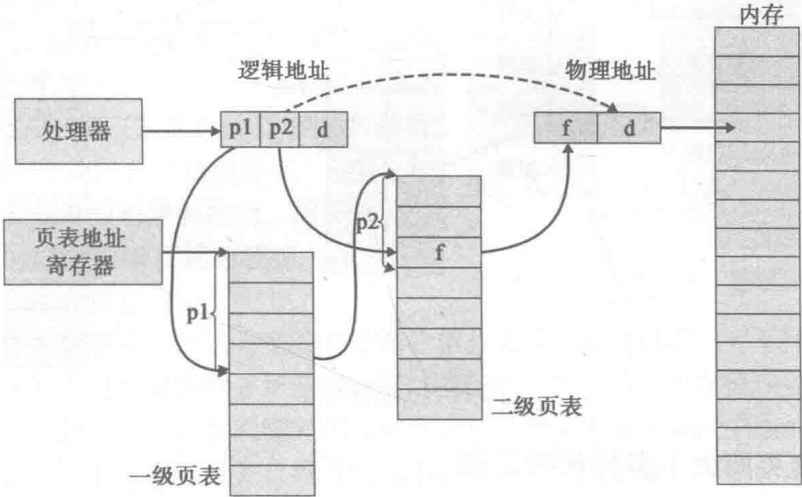


图 11-4 两级页表

231

两级页表被证明是一种非常有用的技术并且还得到了扩展利用。现代处理器通常支持三级或四级页表架构。请注意，这可能会潜在地导致我们的有效的内存访问时间发生变化。对于四级页表来说，在最坏的情况下，我们可能要进行 5 次内存访问才能访问到内存中的真正的所请求访问的那个字节，因为可能每个页表引用都不在快表中。故而，我们的有效的内存访问时间公式就应修正为如下的样子：

$$EAT=A(E+M)+(1-A)(5*M)$$

幸运的是，在大多数时候，我们的快表将会持有那些最后的物理内存引用。所以总的来说，我们只是比单级页表需要承受稍大一点的性能损失。

值得注意的是，这项技术具有创建虚拟页表（virtual page table）的功效。由于地址空间是如此庞大，所以页表通常情况下非常稀疏——其中有好大部分地址空间并没有被真正用到。在这样的情况下，那些较低级别的页表的相应部分在其被实际使用之前，并不需要分配内存和进行填充，从而可以节省大量的表空间以及相关访问所必需的资源。

11.2.5 反置页表

针对页表大而稀疏问题的另一种略有不同的方法是把问题转化一下，即把物理页框映射到逻辑页面中。图 11-5 显示了一种处理页面映射的所谓反置页表（inverted page table）方法。反置页表按物理页框号顺序保存，且该表本身将被搜索以查找相应的引用。鉴于整个系统只有一张这样的表，所以仅提供来自各个进程的页号并不足以指定对应的映射。例如，每个进程都应拥有页号 0。因此，进程标识符必须与页号一起存放在反置页表中。搜索反置页表往往要比搜索普通的页表慢些。一般而言，操作系统可以通过使用散列函数（也称为哈希函数）来访问该表以提高相应搜索速度。由于有些进程可能拥有散列到相同数值的页号 / 进程标识符组合，所以相关方法还需要通过一种链接方案来解决冲突。因此，反置页表甚至相比于普通的页表，使我们更加依赖于通过快表查找来完成我们的大多数查找。尽管如此，反置页表确实比普通页表使用的内存要少得多。

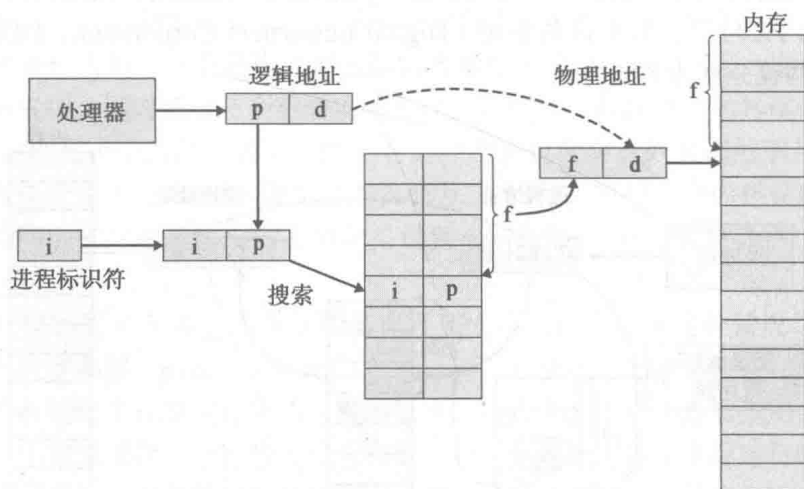


图 11-5 反置页表

11.2.6 支持页面大小多样化的页表

在后来的系统中已经普遍支持两种以上的页表大小。例如，在英特尔奔腾处理器体系结构中，普通的页面大小为 4096 B，但是同时还存在另一种可能的页面大小为 4MB。这样做的理由是为了使操作系统的内核可以在进程的页表中进行映射，但不会让页表为此占用太多的内存。大多数的内核页面在每个进程中都是相同的，并且它们从不移来移去，也不会导致碎片，而且它们始终保持在相同的位置，因此不需要让它们跟具有未知大小和未知持续时间的进程一样，也分解成小的页面。此外，正如我们很快就会看到的那样，在进程逻辑空间的应用程序部分，有关页面有时甚至不在内存中。但这对于操作系统内核来说，往往并非如此，尽管作为内核部分的某些操作系统页面可能也会这样。总之，仅仅通过一个或几个页面来映射整个内核能够使有关设置和操作更加简单方便，并且甚至只需要一个快表项就可以映射整个内核，这是一个很大的优势。

在一些后来的 UltraSPARC（超级可伸缩处理器体系结构）处理器中，有关软件可以为应用程序的不同部分分别选择各种不同的页面大小。我们将会在段页式一节看到相关工作机理的介绍。

11.2.7 历史的注脚

尽管现代系统通常会在并发运行多个进程的环境中使用这些技术，但是在历史上，在仅仅运行单个进程的情况下就曾经出现过使用分页技术的一些系统。程序可以引用程序中那些尚未进入内存的部分，这在很大程度上就像它们在调用上一章曾经讨论过的覆盖例程一样。这具有一个优点，也就是允许支持运行比物理内存大得多的进程。在较小内存的时代，这是一个很大的优点，但其在当前的系统中并没有被大量地利用。现代操作系统往往使用的是请求分页技术，在后面一节将会展开讨论。

11.3 分段

几乎是在分页技术设计的同时，另一种被称为分段 (segmentation) 的技术也被设计出来并沿着一种不同的发展轨迹向前演化。分段技术相关设计主要是为了帮助解决与分页寻址相同的问题，但还包括其他的一些考量。分段技术源自于我们能够把一个程序看作是由若干不同部分组成的观察。我们往往会拥有一个主例程，而且也常常会拥有若干子程序和函数，它们通常被编译器识别为单独的项。有时我们甚至单独编译子例程和函数并将它们放入库中。我们还设立有相关区域用来存放堆栈 (或简称栈)、静态数据项、常量信息、文件缓冲区、通信缓冲区，等等。这些区域都可以各自单独创建和控制。图 11-6 显示了一个程序的分段组成以及其加载到主存之后所构成的进程的程序段集合。

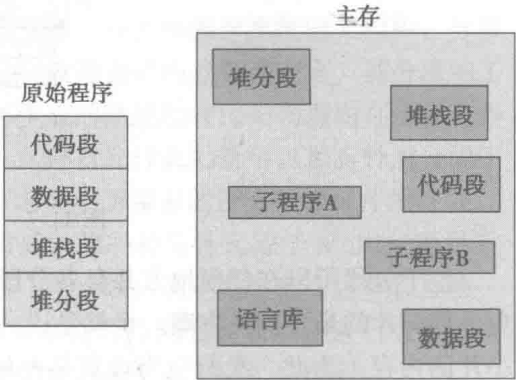


图 11-6 进程分段

其中每一个部分都可以被认为是与其他部分相分离的，并且可以拥有单独的逻辑寻址空间。例如，由于数据具有单独的寻址空间，所以我们应当认为第一个数据项存放在数据段地址空间中的地址 0 处。于是，我们需要相关硬件针对所有关于每个分段中地址的引用进行重新定位，有关方式与利用重定位寄存器针对关于整个进程的引用的重新定位是一样的。具体而言，我们将会使用一种很像页表但又存在些许不同的机制，称之为段表 (segment table)，相关样例如图 11-7 所示。我们仍然认为逻辑地址可以被划分为两个部分，但应分别是段号 (segment number，在图中标记为 s) 和偏移地址 (displacement，在图中标记为 d)。采用分页方式时，我们拥有数量相当多但尺寸却相当小的页面，所以有关偏移地址所占位数不是很多，但页号所占位数较多。而对于分段方式来说，我们则拥有相对较少数量的分段，但每个分段本身则可能相当大，因此段号通常所占位数不是很多，但段内的偏移地址所占位数则较多。此外，页表中的表项包含的是页框号，而段表中的表项包含的则是内存地址。通常情况下，程序设计人员不会对分段施加任何公开的控制。而编译器将会为正在编译的模块的主要部分——程序代码、堆栈、堆、全局变量等——生成单独的分段，并在目标模块中放置上对它们的符号引用。链接器将会分配实际的段号，这些段号在链接器把相关目标模块合并到可执行二进制程序时或者在操作系统实现库模块的动态加载时将会用到。

在图 11-7 中，我们可以看到对包含子程序 A 的分段的一个内存引用。有关硬件将会用到由逻辑地址的段号所指定的段表项。它将会提取对应段表项中所找到的相应分段地址指

针，并将其与逻辑地址的段内偏移地址部分进行相加，从而计算获得对应的物理地址。而支持分页方式的相关硬件只是用页框号把页号替换掉即可，其根据在于页框的大小和页面的大小总是相同的，所以页框和页面也总是沿着块边界来确定位置的。但是，由于分段具有可变的大小，所以它们可以位于任何地方，因此我们应当通过使用段表给出的相应分段指针加上偏移地址来获得物理内存地址。请注意，就像分页方式一样，采用分段方式也会对每次访问产生额外的内存引用开销。故而，采用分段的系统也应当利用快表来提高访问速度。

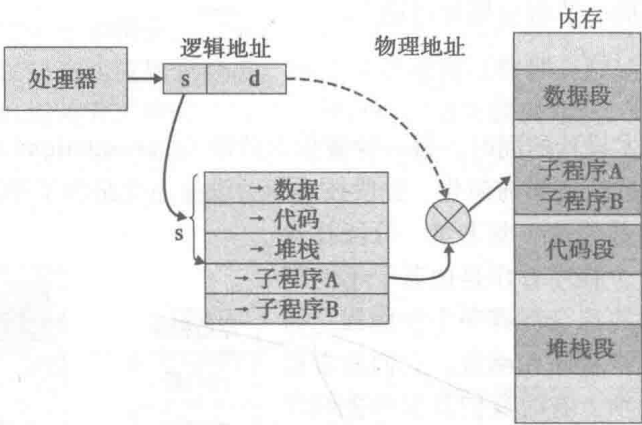


图 11-7 主存中的段表和分段

鉴于分段可以在任何地方并且各分段的大小不是全部一样的，所以仅凭这些尚不构成避免外部碎片的最佳解决方案。也就是说，我们仍然必须得跟踪内存空穴，我们仍然无法分配小片的内存。为此，我们应当设定某种最小颗粒度——譬如说 1024 字节。因此，我们可能将会产生一些内部碎片。但是，现在的空穴大小的范围将会比我们在监测整个进程时所考虑的空穴大小的范围要小些，因为我们把每个进程分解成为（很可能是许多的）分段。总之，与针对整个进程的内存管理相比，我们在外部碎片方面所面临的问题已有所缓解。

由于分段具有可变的大小，所以我们必须得为系统提供一种地址检查的途径，以便我们可以确保进程不会在对应分段的界限之外寻址。每个分段的界限应当与指向该分段的指针一起存放在段表中。正如我们前面围绕分页机制所讨论的那样，鉴于各分段具有不同的目的，我们还可以通过限制对各个分段进行的访问类型来增强我们对系统所提供的保护。通常情况下，每个分段可以设立一组标志位，以用于控制我们可以实施的访问类型。例如，一个数据常量分段可以被标记为只读，程序分段可以被标记为只执行，而堆栈和数据分段应当被标记为允许读和写但不可执行。

235

在一些操作系统中，若干进程共享相同分段是被支持的。例如，我们可能拥有几个用户同时运行一个程序编辑器。为此，我们可以为每个用户创建一个进程，并在各个进程的段表中进行代码段的映射且使它们都指向物理内存的相同部分。如果我们拥有标准语言的通用运行时库，即使是对于不同的程序，我们也可以映射有关分段指向相同的物理内存分段。对于操作系统来说，关于牵涉多个进程的段号的管理往往可能比较繁琐。

通常情况下，用高级语言编写程序的程序员不会意识到被操作系统所使用的分段机制，而只有当他们的程序产生分段错误时才会有所察觉，且相关分段错误大多数情况是由用作堆栈的分段的溢出所引起的。一般来说，编译器和链接器负责通过调用管理段号的操作系统例程来为各个分段分配段号。而同低级语言打交道的程序设计人员则需要了解分段以及操作系

统关于分段的使用方法，并且如果需要，他们还可以控制分段。另外，Windows NT 系列没有使用分段机制，其理由是分段在许多硬件设计上没有需要、在英特尔以外的其他设备上不可用并且使用分段机制会降低软件的可移植性。Linux 仅以有限的方式来使用分段，具体我们将会在下一节中进行讨论。而大数 UNIX 衍生的操作系统则使用段页式内存管理方式，同样也将在下一节展开讨论。

11.4 段页式

分页和分段之间存在着根本区别。分页对运行的进程是透明的。对于一个应用程序来说，如果其原先创建是运行在有关进程将被映射到一个单一内存分区的操作系统中的，那么其有可能不用更改就可以运行在使用分页存储器架构的系统上。另一方面，分段则要求程序以某种方式被结构化，以便它们被划分为具有不同地址空间的若干逻辑部分。关于分页和分段，一个很有意思的方案是，利用适当的硬件设计，我们便可以将分段程序体系结构与分页内存体系结构有机地结合到一起，称之为段页式内存管理。在各种操作系统的文档中，分段可能被称为区域（region）或内存区域（memory area）。就段页式内存管理而言，其分段的工作机理与我们已经描述的分段方式基本一样，只是所生成的地址没有用作物理内存地址，而是被视为逻辑地址并进一步通过分页机制进行转换。这样就允许我们既可以拥有针对分段引用访问方式的精确控制，又可以拥有分页机制固定页面大小所带来的没有外部碎片的好处。

关于分段和分页的结合，一般存在两种不同的方式。第一种设计起源于 Multics 项目^①。在这种设计方案中，我们将会为进程的每个分段配备一个页表，而不是为整个进程设立唯一的一个页表。相关设计如图 11-8 所示。首先，根据地址的分段部分在段表中进行查找，并返回一个指向页表的指针，该页表实现了分段内页号到物理内存中页框号的映射。

236

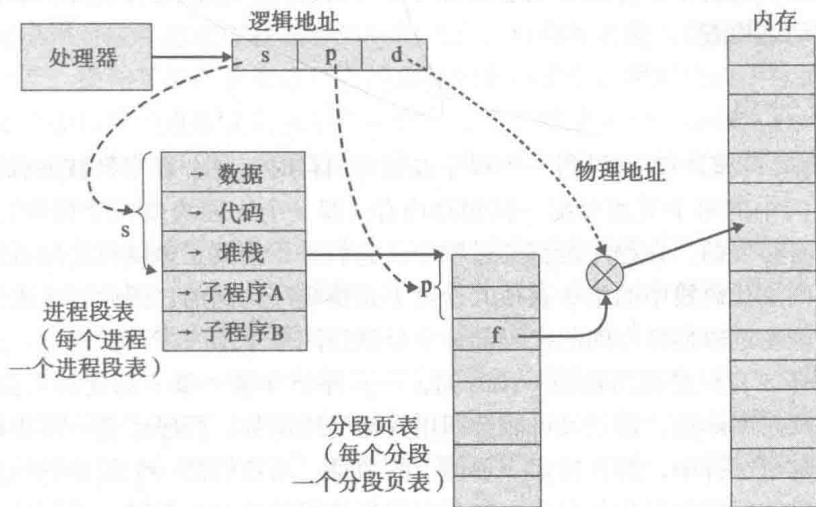


图 11-8 传统段页式内存管理

第二种设计方案被用于更现代的系统。对于这种设计方案而言，仍然有一个段表，但其不是指向每个分段的单独页表。换句话说，段表中的地址在于一个线性地址空间范围内，其将会以与分页式系统相同的方式映射到物理内存中。相关设计如图 11-9 所示。在这种情

^① <http://www.multicians.org/fjcc1.html>

况下，段表项描述了线性地址空间的一部分，而这部分线性地址空间可以被看作是对应于相应分段的页表。但是，就有关硬件而言，它只是单一页表的一部分。

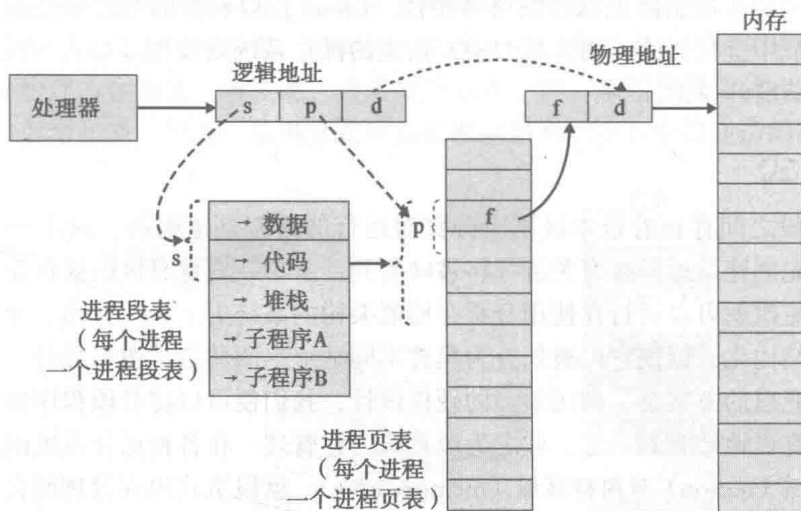


图 11-9 现代段页式内存管理（或称为线性寻址式分段内存管理）

大多数现代操作系统以这样或那样的形式采用了第二种机制，但是，它们往往限制了有关分段的使用。例如，在 Linux 系统中，除了有一个分段用来描述整个运行进程外，仅仅将分段机制用于内核的处理。同时，有关分段经常被用来限制寻址和控制访问。例如，两个分段可同时被用于映射到相同的内核地址空间。其中一个用于程序的执行，故而可被设置为允许执行但不能读取或写入。另一个则用于数据的访问，故而可被设置为允许读取和写入但不允许执行。另外，还有一个分段可用于运行时堆栈的访问，以便允许硬件机制能够有效地进行堆栈溢出的动态检查。

11.5 请求分页

到目前为止，我们一直假设当一个程序被装入内存时，整个程序将被加载到内存并且会为逻辑寻址空间中的每个页面分配一帧物理内存（即一个物理块或一个页框）。然而，最终意识到这是没有必要的。在程序运行的过程中，它们并不会真正地随机访问地址而遍及其整个逻辑地址空间。代码段中的指令在很大程度上是被顺序访问的，因此对于大约一千条指令而言，我们可能在逻辑地址空间的代码部分中也就访问单个的一个页面而已。或者程序可能会进入一个循环，有时是相当长的一段时间，一直停留在某个单一的代码页面上面。诚然，我们会频繁地调用库函数，而这又可能会调用其他的库函数。程序还会一步步地处理一个数组，或者扫描一个字符串，或者搜索一条传入的消息。当我们把一个程序的执行过程划分成一系列小的时段并仔细查看在各时段中内存引用所访问的相应页面时，我们往往会发现在任意给定的时段中有关进程仅仅访问到了有限的几个页面。这种现象在操作系统设计中非常重要，被称为引用的局部性（locality of reference）。我们在缓存机制以及我们许多其他的操作系统算法中都用到了这一相同的理念。

利用这种现象而开发的相关技术被称为请求分页（demand paging，或称为请求调页）。其想法是，我们应当对页表项中的有效位的含义稍微进行一下调整，而与该有效位的使用相关联的硬件无须进行任何改变。进一步说，该有效位用来标示对应页面尚未分配页框。在我

们以前的设计中，有效位则意味着对应页面位于当前程序的逻辑地址空间之外。现在，有效位仍然可以用来标示这一含义，不过也有可能仅仅指示当前没有物理页框被映射到对应页面。当我们加载第一个页面时，我们将会把其有效位设置为真，以指示该页面已在内存中。同时，我们将会把其他的每一个页面的有效位设置为用以标示对应页面不在内存中的情形。然后，我们将启动程序的运行。从理论上来说，我们甚至不用把任何页面加载进入物理内存就可以启动一个程序的执行。操作系统可能只是通过转向执行内存中当前进程的方式，就触发了页面故障机制，进而把对应程序的第一页装入内存。这被称为延迟加载（lazy loading）。即便我们加载装入了程序的第一个页面，但它很快将会引用到尚未进入内存的另一个页面中的数据。图 11-10 就显示了此类页表的一个示例。于是，有关引用将会获取页表项，而该页表项有效位的设置将会触发一个“内存寻址错误”的中断。在这种情况下，内存管理子系统将会查看对应引用，以确认该引用是否引用了一个确实是在当前程序的逻辑地址空间但尚未装入内存的页面。而如果有关引用并不是引用了当前进程的逻辑地址空间中的一个页面，则说明对应程序犯了一个错误，应抛出一个寻址异常，很有可能会中止对应进程。

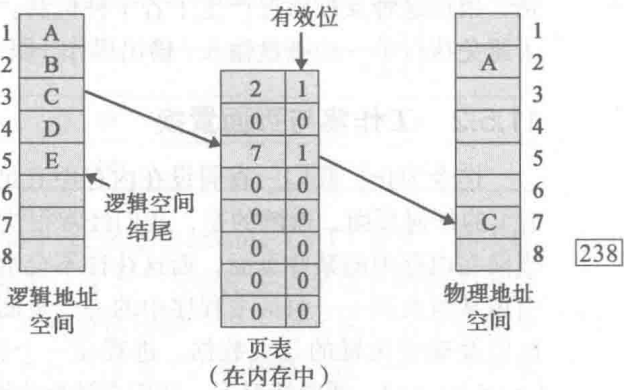


图 11-10 请求分页式页表示例

如果引发故障的地址是在进程的逻辑地址空间中，那么相应页面仅仅是当前未被映射到物理内存而已，或者是因为其从未被装入内存，或者是已经从内存中移除。这种条件被称为页面故障（page fault，或称为缺页）。于是，内存管理子系统将会请求从辅助存储器读取对应页面，同时操作系统将把相应程序置为等待状态，直到相关读入操作完成为止。一旦对应页面被读入内存，操作系统将会更新页表使其指向新的页框，将对应表项标记为有效，并且重新启动产生页面故障的进程使其从触发故障的指令处继续运行。请注意，这一机制对应用程序仍然是透明的。换句话说，应用程序设计人员通常不需要知道有关进程的执行状况，更不用说采取任何操作措施了。

在某些情况下，这将允许我们运行那些非常庞大以至于内存空间无法容纳下的程序。“80/20 规则”通常是成立的——一个程序 80% 的代码是编写用来处理只会在 20% 的情况下发生的事情。因此，在许多情况下，有关程序 80% 部分中的大部分将永远不会被加载到内存中。作为一个额外的好处，这将允许我们的程序启动得更加快捷，因为如果一个页面从未引用过，我们就不用把它加载到内存中。同样，在我们试图运行许多程序的环境中，或许可能是多用户环境，在给定的物理内存容量的前提条件下，总体而言我们将能够同时运行更多的程序。

11.5.1 请求分页方式下的有效内存访问时间

或许你还记得，当我们第一次评判分页机制时，我们曾领会到页表使用本身是如何让有效的内存访问时间翻倍的。这便需要引入快表机制来缓存工作集中的相关页面引用的页框号。现在考虑一下，当我们访问一个不在内存中的页面时，有关访问时间将会发生什么样的变化。我们的有效访问时间将会包括四方面的组成要素，如表 11-1 所示。（表中所显示的速

[239] 度仅仅是近似的相对速度，而不是具体的期望值。)

表 11-1 请求分页有效寻址时间

要 素	相 对 速 度	要 素	相 对 速 度
快表查找	1ns	磁盘写（脏页）	20ms
内存访问	100ns	磁盘读取	20ms

如表所示，相对而言，磁盘输入 / 输出所花费的时间大大超过了内存的访问时间。于是，根据这种支配状况产生了若干种机制。有关机制乍一看起来极其复杂，但它们仅仅是为了避免执行单一的磁盘输入 / 输出操作（服务于请求分页）而开发的。

11.5.2 工作集与页面置换

迄今为止，我们一直假设在内存中有足够的空闲页面可以加载我们所需（即当进程引用时）的任何页面。遗憾的是，我们经常很快就会用完空闲页面。在这时候，我们便需要移除当前在内存中的某些页面，而这往往不会给我们带来任何问题。当一个程序运行时，它将会引用某组页面——包括主程序中的一个页面，还有或许是几页的库例程、输入和输出的缓冲区以及确定无疑的几页数据。进程在一个较短的时间段内所引用的这组页面被称为工作集（working set）。通常情况下，我们在被称为滑动窗口（sliding window）的某个固定时间间隔上测度工作集。

例如，假设某进程拥有包含标记为 1~7 的 7 个页面的逻辑地址空间，并且关于这些页面的引用序列如下所示：

1 2 1 5 7 1 6 3 7 1 6 4 2 7

我们将通过查看最近的 4 个引用来监测相应的工作集。伴随这个引用序列的逐渐展开，有关的工作集将会发生改变，如表 11-2 所示。在进程运行的过程中，有关工作集通常会随时间而发生变化。特别地，发现一个进程拥有在内存的若干页面但不再被引用，这是很正常的。在上面的页面引用序列中，我们可以看到第 5 个页面在步骤 4 之后就没有再被引用过，所以我们可以从内存移除该页面。我们想要做的就是识别这样的页面，并在我们不再需要它们的时候把它们从内存中移除出去。（把我们所认为的可能不再需要的页面移除出去称为页面替换、页面置换或页面淘汰。）但令人遗憾的是，我们往往无法真正做到此类识别。我们在一段时间内没有引用一个页面，并不意味着下一条指令就一定不会引用这个页面。在上面的页面引用序列中，我们可以看到第 2 个页面在步骤 2 和步骤 13 之间没有被引用过，故而我们不再把它看作是在工作集中（如表 11-2 所示），因为我们查看的仅仅是一个 4 步的滑动窗口。幸运的是，从内存中移除一个后来还会需要的页面并不会造成任何破坏。其导致的结果仅仅只是不那么高效而已。具体而言，针对相应页面的下一次引用将会导致一个页面故障，从而使该页面被再次提取。

[240] 有一种非常简单的页面淘汰策略，即先进先出（First In First Out, FIFO）淘汰算法。在该算法中，操作系统需要为每个进程保存一张根据相应页面加载进入内存的次序而形成的页号队列，并在需要时强制移除最早进入内存的相关页面。这种算法是一种开销较低的算法，只需要操作系统付出很少的开销。然而，尽管先进先出算法成本较低且易于理解和实现，但是它性能差且不太稳定，因此如今很少被人利用。这种算法曾在 VAX/VMS 操作系统中使用

过。此外，该算法被检测证实会发生贝莱迪异常现象（译者注：Belady’s anomaly，即缺页率会随着所分配得到的页框数的增多而增大的现象）。

表 11-2 工作集监测示例

事件编号	工 作 集	事件编号	工 作 集
1	1	8	1 3 6 7
2	1 2	9	1 3 6 7
3	1 2	10	1 3 6 7
4	1 2 5	11	1 3 6 7
5	1 2 5 7	12	1 4 6 7
6	1 2 5 7	13	1 2 4 6
7	1 5 6 7	14	2 4 6 7

理论上而言，存在一种最佳页面淘汰（Optimal Page Replacement, OPT）算法。其工作机理如下：当需要淘汰某个页面时，操作系统将会把对应下一次使用是将来最长时间的那个页面淘汰出去。例如，相对于从现在开始在 10ms 时将被使用的一个页面来说，那个从现在开始在 200ms 时才会被使用的页面将会被优先选中和予以淘汰。这种算法在一般情况下是无法使用的，因为，除非是在极其限定的情况下，否则根本不可能知道多长时间一个页面将会被使用。不过，如果这种算法可以实现，那么它将会是我们能够做到的最好的算法，所以值得讨论一番。假设我们拥有仅仅 3 个空闲的页框来运行一个进程，如果我们针对前面所给出的页面引用序列来使用最佳页面淘汰算法，那么，将会产生 9 次页面故障（即缺页中断），包括加载前 3 个页面所对应的相应页面故障。

还有若干种其他的机制我们也可以用来选择要淘汰的页面。一种方案是试图推算出哪个页面是最长时间未被引用的。正如通常所说的，这个页面就是最近最久未使用的（Least Recently Used, LRU）那个页面。我们将做这样的假设，这个页面就是最有可能不再被使用的那个页面，故而我们将把它从内存中移除出去。如果我们试图实际来保存每个页面的最后一次引用的时间，我们最终将需为每个实际的内存引用创建至少一个额外的内存引用。因此，实际的操作系统并不直接实现最近最久未使用淘汰算法。然而，借助于某些硬件功能，我们可以识别某段时间内没有被使用的有关页面。利用相关功能的最简单的算法称为时钟算法[⊖]（clock algorithm）。仅仅付出相当低的代价（就额外的内存引用而言），有关硬件便可以确保在一个页面被引用时，页表项中的某个二进制位被置为 1。这一位通常被称为页面引用位（page reference bit），有时也被称为存取位（access bit，或称为访问位）或使用位（use bit）。（如图 11-11 所示。）当一个页面（通过页表）被引用时，有关硬件将会检查对应页表项中的页面引用位。如果它已经被设置为 1，那么无须进行任何操作。如果它没有被设置为 1，那么就将它设置为 1，这可能会花费一个额外的内存周期的开销。有时，操作系统可能会针对当前在内存中的相关页执行这些位的清零操作。我们把页表中的所有页面的有关位进行清零，让对

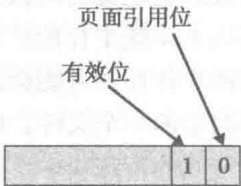


图 11-11 页面引用位与有效位

241

⊖ 这个未必恰当或者说算不上成功的术语并没有参考系统时钟。它只是指这样的设想，即当操作系统到达页面引用列表末尾的时候，就会从头开始再来，这非常像一个时钟掠过 12 的时候再返回 1 继续的情况。

应进程运行一段时间。在此期间，有关硬件将会为所有被引用的页面的对应页表项的页面引用位进行设置操作。当我们需要查找一个页面从内存中移除的时候，我们将会搜索相关表格，并找到一个有效位为 1 但引用位为 0 的页面。这样的页面将是被淘汰页面的一个很好的候选页面。

我们也可以对这种机制稍微进行一下加强。对于每一个页面，我们可以保存关于页面引用位相关设置以往历史的一个字节或者更多字节的信息，称为引用计数（reference count）。这种机制有时被称为老龄化（aging）技术。在我们周期性地对页面引用位进行清零操作时，我们首先将引用计数右移一位，然后把对应页面引用位的最新取值移到引用计数的最高位上。如果某个页面在上一个周期中曾被引用过，那么对应计数将因而具有高的取值。伴随其间该页面未被引用的刷新周期的经过，有关移位操作将会实际上对相应引用计数执行每次整除 2 的操作，所以引用计数会变得越来越小。图 11-12 显示了一个页面的引用计数示例，在最近的两个刷新周期中，移入高位的二进制位是零，因此每次这个引用计数就会变得更小。当我们需要淘汰一个页面时，我们优先选择具有最小引用计数的页面。综上所述，单一的引用位只能用来标明最近的时间间隔中对应的页面是否被引用过，而通过引用计数，则可以使我们更好地了解相关页面在最近一段时间的使用历史。

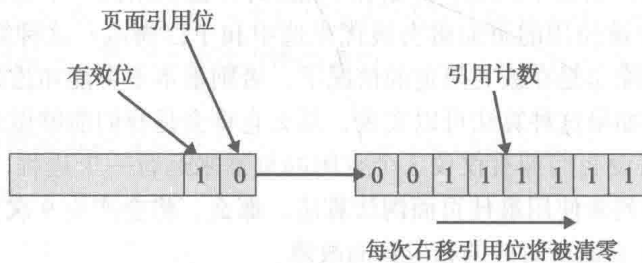


图 11-12 页表项及所关联的引用计数示例

11.5.3 脏页

对于一道程序的代码部分加载到某个内存页面且后来我们又将其淘汰和用别的什么东西来替换对应页面的情况，我们并不需要保存该页面到任何地方，因为如果相关代码部分被再次引用，那么我们可以直接找到原始程序并获得相应的页面。这也正是为什么程序不应该在其运行时修改自身的诸多原因之一。但是，如果一个页面包含有数据，并且自从该页面加载到内存以来，其中的某些内容又发生了更改，那么我们不能只是仅仅替换页面就完事的——我们必须保存该页面中的当前数据，以备其被再次引用。我们把修改过的页面称为脏页（dirty page，或称为脏页面）。我们将会把那些脏页面写到辅助存储器中的特定的地方，有关位置被形象地称为对换文件（swap file）或后备存储器（backing store）。这个对换文件也因而充当了系统主存的扩展空间。这也是术语虚拟内存（virtual memory）的起源。一般而言，该对换文件往往可能会比主存大许多倍。在某些操作系统中，该对换文件的确是常规文件空间中的一个文件，但在其他的一些操作系统中，该对换文件则位于常规文件系统之外的磁盘空间的特殊块中。无论哪一种情况，这种对换文件都是由相应的操作系统以一种特殊的方式来进行访问的，也就是说，有关访问没有经过普通的文件系统接口，而是采用了原始模式的输入/输出例程。此外，对于某些操作系统来说，只有一个这样的文件，但对于另外的一些操作系统来说，则可能把若干单独的对换文件存放到不同的驱动器上以提高系统性能。

11.5.4 其他的页面淘汰算法

现代操作系统使用各种各样的算法来试图优化页面淘汰过程。其中有一种算法称为二次机会算法 (second chance algorithm)。这种算法是在时钟算法基础上修改而形成的, 它与我们首次关于时钟算法所描述的方式来检查页表, 以查找未被引用的一个页面。但是, 当利用这种算法检查每个页面时, 如果它发现相应引用位设置为 1, 则做清零操作。通过这种方式, 这种算法就更新了引用位, 使其来进一步反映比最近的引用刷新周期更晚的一段时间的引用情况。伴随这种算法在页表中的前移查找, 如果其在第一轮查找过程中没有找到任何空闲的页面, 那么它应当在第二轮查找过程中能够找到某个空闲的页面。在某些操作系统中, 这种页面搜索过程是由后台进程而不是页面置换进程来完成的。如果系统中的空闲内存空间非常少, 那么相对于可用内存比较充裕的情况, 操作系统将会更为频繁地运行这个后台进程并且往往会运行较长的时间。而如果空闲内存不是时下引起关注的原因所在, 那么该后台进程的运行频率将会比较低下并且运行时间较为短暂。有关后台 (background) 操作是当没有高优先级进程处于就绪状态的情况下所执行的琐事。也就是说, 在后台任务中所执行的指令不会以牺牲任何用户进程为代价来加以执行, 因此这些指令或多或少是免费的或者说是没有开销的。

值得注意的是, 淘汰一个脏页的代价是淘汰只读页面或干净页面的代价的两倍。这是因为其间操作系统必须对磁盘进行两次访问, 而磁盘访问大约比主存访问慢 1~10 000 倍。因此, 我们可以承受消耗大量的处理器周期以试图找出在当时所知相关信息的前提条件下可淘汰的“最佳”页面。我们可以看到, 此类以耗费处理资源来节省输入/输出的一种方法是通过结合使用脏位 (又称为修改位) 与引用位来改进二次机会算法。这种算法有时称为增强型二次机会算法 (enhanced second chance algorithm, 又称为改进型时钟算法), 有时也称为最近未使用 (Not Recently Used, NRU; 或者 Not Used Recently, NUR) 算法。在这种算法里, 我们根据这两位的设置把相关页面划分为 4 种类型: 1) 干净且未引用的页面; 2) 脏的但未引用的页面 (引用位在对应页面被修改之后的某个时间被清零); 3) 干净但引用的页面; 4) 脏的且引用的页面。具体而言, 我们首先顺次查看页表的相关表项, 以查找第一种类型的页面, 即未引用且干净的页面。如果找到此类页面, 我们可以立即使用。而如果我们没有找到第一种类型的任何页面, 那么我们将再次查看页表的相关表项, 以查找第二种类型的页面, 以此类推。直到第四轮查找, 我们保证可以找到某个页面作为要淘汰的页面, 不过我们常常在此之前便会找到一个更好的淘汰页面。

在请求分页系统中还会出现另一个问题是, 操作系统应当如何选择进程, 以从中提取欲淘汰的页面。存在两种可能, 其一是操作系统只能从引发故障的进程内部选择页面 (局部置换), 其二是操作系统可以从任何进程那里选择页面 (全局置换)。我们经常希望程序设计人员所编写的程序使用的资源最少。而如果程序设计人员编写的程序所产生的页面故障较少, 那么他的程序应该会运行得更快。采用局部置换策略, 一道表现糟糕的程序最多也就伤及其自身。而使用全局置换策略, 一道写得不好的程序可能会拥有一个过大的工作空间, 故而会伤害到其他进程, 进而产生特别多的页面故障。其结果是, 一道精心设计且所产生页面故障较少的程序可能会受到另一道设计不太精良且所产生页面故障较多的程序的不利影响。通过设立一个后台进程来运行二次机会算法以识别可疑页面, 通常可以和全局置换策略协同工作且效果良好。一般来说, UNIX 和相关系统往往采用全局置换策略, 而 Windows NT 系

列操作系统则常常采用局部置换策略。

鉴于内存和硬盘的动态特性，页面置换算法目前仍是一个有大量的研究正在开展的活跃领域。伴随相关尺寸、速度和成本的变化，折中权衡亦在发生着变化，不同的算法往往可能适用于不同的情况和具有较好的效用。

11.5.5 每个进程应当拥有多少页面

当一个操作系统按请求分页方式进行设计时，我们并不会让有关程序在内存中无限地增长。一方面，正如我们在讨论工作集的概念时所看到的，最终在内存中将会有一些相应程序不再引用的页面。同时还有一些其他的页面，在一段时间不会需要，但是我们现在可以让另一个进程更有成效地来使用这些内存，而当我们那些页面时再重新加载它们。为此，便出现了每个进程应该允许使用多少页面的问题。通常可以使用各种不同的方案来设置此类限值。首先，存在某个最低限值设定，即我们不希望一个程序的页面数降至该限值以下。例如，在一些机器上常见的指令类型是内存到内存的操作。就这种情况而言，对应指令本身可能会跨越页面边界，故而我们需要两个页面来访问相应的指令。同时，源操作数（source operand）和目标操作数（target operand）也都可能跨越页面边界，因而对于单个进程来讲，在这种类型的机器上存在 6 个页面的绝对下限。即使是有这样的设定，一个程序却往往可能拥有一个比此绝对下限值要大的工作集。但是，什么才是一个合理的上限呢？

[244]

我们可以在一个原型系统上研究程序的运行，并随意设置某个限值。但是，如果没有足够多的进程正在运行并且相应页面并未填满所有可用的内存，那么我们将会产生一些我们本来不必要产生的页面故障。因此，随意设置限值并不是一个很好的主意。我们可以让系统稍微有点动态适应的味道，也就是说，可以简单地通过将可用页面总数除以正在运行的进程数来实施相关分配。这种机制被称为公平分配（equal allocation，或称为平均分配）。不过，这种方案通常也并不合理。进一步说，如果其中一个进程是基于图形化用户界面的计算器而另一个进程是万维网服务器，那么我们可能会合理地推断出网站服务器应当可以使用额外的页面以便获得更大的效益。比较程序的大小是猜测哪些程序可以使用更多页面的一种简单方法。从磁盘空间占用情况来看，网站服务器程序可能会比计算器程序大 100 倍，因此按 100 : 1 的方式来进行网站服务器和计算器的页面分配应当是合理的。这种机制被称为按比例分配（proportional allocation）。不过，这仍然算不上是一种完美的解决方案。考虑可能打开一个小的备忘录文件亦或打开一整本书的字处理程序。显然，打开整本书应当比打开一个小的备忘录文件更可能有效地利用较多的页面。我们真正想要做的应当是拥有某种机制，按照进程对页面使用的比例来进行页面的分配。

11.5.6 页面限值自动平衡

大多数现代的操作系统常常会使用页面限值自动平衡这样的机制，且大多数相关机制是建立在页面故障频率（Page Fault Frequency, PFF）算法基础上的变种。一般来讲，它们依赖于这样的思想，也就是说，一个进程的缺页率（或称为页面故障率）是对应进程是否拥有适当页数的一种很好的指标。如果相应进程拥有的页面太少，那么缺页率将会快速上升。而如果一个进程不产生任何页面故障，那么该进程也可能拥有内存中的一些页面但其并不需要。为此，相关机制设定了关于缺页率的一个上限和一个下限。图 11-13 显示了这种机制的工作机理。如果一个进程的缺页率低于对应下限值，那么操作系统应当从该进程的最大页框

计数值中减去 1。如果缺页率超过了对应的上限值，那么操作系统应当对该进程的最大页框计数值加 1。这种机制将倾向于保持系统中的所有进程在同样缺页率的情形下运行，并且将会向一个进程分配其需要保持在相应范围的尽可能多的页框。

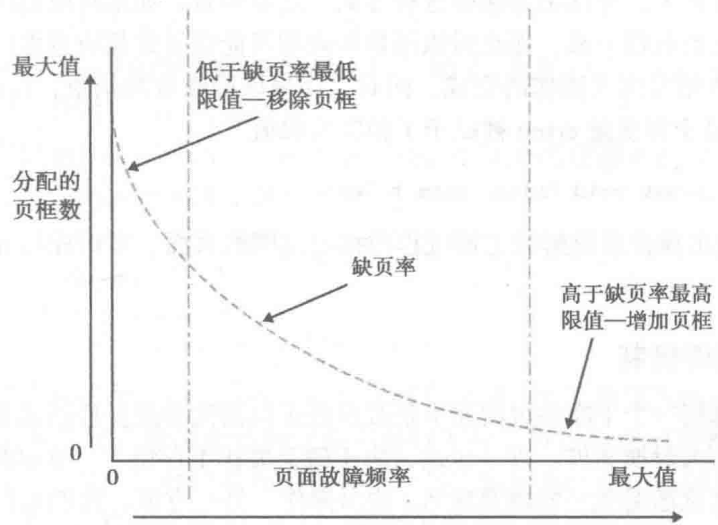


图 11-13 页面限值自动平衡

11.5.7 抖动

假设现在我们针对一个进程可以使用的页数设置了一个硬上限——不妨称之为限值 N 。进一步假设，这个进程的设计是这样的，也就是说该进程已经到达其执行的某一阶段且其工作集超过了 N 页的上限。最后假设我们仅仅使用了局部页面置换策略，故而当有关进程产生一个页面故障时，我们将会把该进程已经映射的一个页面淘汰掉以进行置换。于是，这一进程将会不断地产生新的页面故障，并将其大部分时间都花费在等待磁盘输入/输出操作方面。其结果是，该进程完成的实际工作很少，而系统还会遭受到过多的磁盘输入/输出操作压力。这种现象被称为抖动（thrashing，或称为颠簸）。在这个例子里，只有唯一的一个进程发生了抖动。需要指出的是，抖动并不取决于我们在这里设想的那些限制条件。如果所有正在运行的进程的工作集的总和大于实际的主存，那么整个系统将会花费较多的时间来进行页面的淘汰和替换，而不是把有效的时间花在运行进程方面，这时我们会说整个系统发生了抖动。一旦发生了抖动，则往往很难停下来，因为恰恰是那些用来终止某些可能并非必要的进程的执行操作行为本身，将会把更多的代码加载到内存，并实际上可能会使情况变得更为糟糕。

245

11.5.8 页面锁定

主存通常会用作输入和输出操作的缓冲区。如果一个缓冲区页面存在尚未完成的输入/输出操作，那么该页面有可能当前未被应用程序修改，故而可能最终会被分页机制选择来重新使用——这显然会带来灾难性的结果。为了防止这种不幸的事件，支持请求分页的操作系统必须允许一个应用程序（往往是设备驱动程序）锁定（lock）一个页面，以便分页机制不会选择对应页面。以下是源自 POSIX 规范的关于这些功能的典型的函数原型：

```
int mlock (const void *addr, size_t len)
```

该例程请求操作系统锁定来自主调进程逻辑地址空间的指定范围的页面。要锁定的页面范围从地址 `addr` 开始且大小为 `len` 字节。只有完整的页面才能够被锁定，故而该范围实际上包括了任何包含指定地址范围的任何部分的页面。如果该函数成功返回，那么每个页面都会绑定到一个物理页框上，并标记为保持这种方式。这意味着，如果对应范围中的某些页面当前尚未加载到内存和驻留下来，那么对该函数的调用可能会导致相应页面的调入，并且该函数将会阻塞和等待相关调入操作的完成。同时，如果该函数成功结束，返回值为零。否则，返回值为 `-1`，并且全局变量 `errno` 被赋予了相应的取值。

246

```
int munlock (const void *addr, size_t len)
```

`munlock` 例程则请求操作系统解锁主调进程的指定范围的页面。本例程与 `mlock` 的功能刚好相反。

11.5.9 页面清理机制

如前所述，选用一个干净的页面而不是脏页面予以淘汰和进行替换是非常重要的。这样就无须把脏页面写入对换文件。进一步说，由于磁盘要比主存慢 $1 \sim 10\,000$ 倍，所以我们可以花费很多指令来试图避免一次磁盘输入/输出操作。另一方面，我们可以尽量在后台完成一些磁盘操作，而不是在我们等待一个页面被加载的时候来做磁盘操作。

我们可以通过保持一个干净的空闲页框池的可用性来减少使用脏页面所带来的影响。当页面淘汰算法选择一个脏页面作为替换的牺牲品（即淘汰页）时，操作系统就可以使用来自空闲页框池的一个干净的页框。然后，在后台将脏页的内容写到磁盘上。而当脏页变得干净的情况下，则可以把相应的页框放到空闲页框池中。

可以在后台完成的另一项任务是清理脏页。具体而言，可以设立一项后台任务来查找最近未被引用的页面（故而可能会作为候选的淘汰页面）并且是脏的页面。同时针对这些页面启动执行一项后台写操作任务，从而使将来选择这些页面作为淘汰页面进行替换时，它们是干净的。当然，有关页面可能会在对应进程运行时再次变脏，但是我们是在后台以最低优先级来完成此项工作的，因此，我们不会浪费可能要做其他事情的输入/输出周期或处理器周期。换句话说，我们往往利用的是无其他正事可做的输入/输出周期或处理器周期来完成此类后台任务。

11.5.10 程序设计与缺页率

一般来讲，我们认为虚拟内存和请求调页（或称为请求分页）对于应用程序来说是透明的。但是，确实存在关于程序设计如何影响缺页率的一些评论。例如，考虑一张比较大的表的搜索问题。假设相应表足够大，以至于会跨越多个页面，并且该表将被多次搜索以查找不同的项。采用二分搜索，我们将在每次开始搜索时碰上中间的页面。然后，我们将可能会碰到另外的两个页面中的一个页面，或者在前半部分，或者在后半部分。于是，这三个页面至少可能会在大部分的时间驻留在内存中，这样我们在这些页面上就很少会出现页面故障（即缺页问题）。然而，对于散列表搜索而言，几乎每次查找都会导致不同页面的读取，因为散列表（或称为哈希表）的基本目的是随机地寻址整个表，以期通过第一次引用就能够撞上所查找的表项。因此，非常庞大的散列表往往不能在虚拟内存系统（或称为虚拟存储系统）中很好地运作。

接下来,考虑执行矩阵乘法的某个程序的如下部分[⊖]:

```
for(i=0;i<500;i++)
    for(j=0;j<500;j++)
        for(k=0;k<500;k++)
            x[i][j]=x[i][j]+y[i][k]*z[k][j];
```

[247]

当这段代码在配置为 MIPS R4000 处理器和 1MB 高速缓存的 Silicon Graphics 系统上按双精度浮点数数组运行时,其运行时间为 77.2 s。

然而,我们可以就这段代码做一个小小的变动,也就是改变循环的顺序,使最内层的循环一步步通过同一页面的内存区域,如下所示:

```
for(k=0;k<500;k++)
    for(j=0;j<500;j++)
        for(i=0;i<500;i++)
            x[i][j]=x[i][j]+y[i][k]*z[k][j];
```

上面第一个例子的问题在于,有关数组是存放在内存中的,故而相邻行的元素(第一个下标)是连续的。但由于其控制最内层循环的变量不是行下标,所以每次引用都将指向不同的页面。而当我们像第二个例子那样改变相关循环后,每次循环都将引用相同的页面,于是由于页面故障数量(即缺页次数)的减少,运行时间便下降到了 44.2 s。

综上所述,一般来讲,虚拟内存和请求调页的操作对于应用程序是透明的,这意味着程序设计人员无须太多地关注这些机制——相应代码无论以哪一种格式都可以正确地工作。但正如我们刚才所看到的,这并不意味着虚拟内存和请求调页的相关操作在任何情况下都不会产生影响。

11.6 特殊的内存管理主题

11.6.1 进程间内存共享

分段和分页都允许在进程之间共享部分的内存空间,而且这可以大量地节省内存。例如,在支持多用户的大型机上,可能经常会见到许多用户同时运行一个字处理程序。通过分页机制,许多进程的页表可以都指向内存中的相同页框,这样在内存中实际上只驻留对应程序代码的一个副本即可。类似地,通过分段机制,许多进程的段表可以都指向相同的物理内存分段而达到同样的目的。虽然这样非常方便,但也可能会带来一些问题。进一步说,如果被共享的内存部分是数据分段,那么各个进程将会更改某些页面。而这在某些情况下可能确实期望如此,但也可能在另外一些情况下并非所愿。若干进程可能会使用共享内存存在它们之间进行通信。在这种情况下,我们希望每个进程都能够看到对页面的所有更改,因此有关进程应该查看物理内存中的相同页框。但是,考虑一下进程创建子进程的情形。一开始,理想的状况应当是在两个进程之间共享整个物理地址空间。但是,当这两个进程运行时,一个进程所做的更改却往往不应该被其他的进程所看到。为了满足这种现实需求,操作系统可以使用一种被称为写时复制(copy on write,或称为写时拷贝)的机制。于是,一开始,这两个进程将被映射到相同的物理页框上,但相应的页表(或段表)的表项将被设置为只读模式。如果其中有任一进程试图写操作这部分共享内存,那么就会发生一个中断。而当这种情况发

[248]

⊖ Patterson, David A. and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2004, p. 617.

生时，内存管理子系统将为每个进程分别创建共享内存部分的单独副本，并从对应的表项中移除写保护标志，同时调整为每个进程只能允许查看其自身版本的数据。

Solaris 系统支持使用 `shmget()` 例程来访问共享内存块（Solaris 称之为分段）。一个进程可以通过该例程的第一次调用来创建一个共享块，而该共享块则由具有指向一片物理内存区域的唯一标识符的控制结构来加以描述。该共享块的标识符一般称为共享内存标识符（简记作 `shmid`）。

下面是 Solaris 系统用于访问共享内存块的调用说明：

```
int shmget (key_t key, size_t size, int shmflg);
```

参数 `key` 或者是 `key_t` 类型的一个变量，或者是常量宏 `IPC_PRIVATE`。它是用来赋值给所返回的共享内存块的数值键值。参数 `size` 是所请求的块的大小（以字节为单位）。参数 `shmflg` 用来指定初始的访问权限和创建时的控制标志。

如果相关调用成功，它将返回一个用来标识共享内存块的标识符。另外，这一调用也可用于获取由另一进程创建的已有共享块的标识符。

以下代码举例说明了 `shmget()` 的基本用法：

```
key_t key;           /* 用作传递给函数 shmget() 的键值的参数 */
int shmflg;          /* 用作传递给函数 shmget() 的初始访问权限和创建时控制标志的参数 */
int shmid;           /* 用于接收函数 shmget() 的返回值 */
int size;            /* 用作传递给函数 shmget() 的块大小的参数 */
shmid = shmget(IPC_PRIVATE, size, shmflg);
if (shmid < 0) {
    printf("shmget error\n");
    exit(1);
}
```

服务器和客户端可以采用 `fork` 调用的方式进行创建，也可以互不相关。对于子进程而言，如果一个共享内存块是在创建相应子进程之前请求和附着上去的，那么服务器可能想使用 `IPC_PRIVATE`，因为对应子进程拥有服务器的地址空间的副本，其中就包含有所附着的共享块。但是，如果服务器和客户端是相对独立的进程，那么使用 `IPC_PRIVATE` 并不是一种好的方案，因为客户端并不知道对应的键值。

11.6.2 内存映射文件

大多数的现代操作系统都支持一种特殊的称为内存映射文件（memory mapped file）的内存共享模式。在这种模式中，一个进程可以请求操作系统打开一个文件并将该文件中的全部或部分数据与该进程的逻辑地址空间的某个区域相关联。然后，该进程就可以将对应空间区域中的信息作为数组或通过内存指针来进行引用。这种系统具有两个主要的优点。第一个优点是相应进程不必使用输入/输出语句来访问有关数据——请求分页系统负责访问对应文件来存取合适的数据。第二个优点是两个或多个进程可以同时请求操作系统访问同一个文件。相同的内存页框将被映射到相应两个或多个进程的逻辑地址空间中，支持它们对内存的共享访问。因此，这种机制为两个或多个进程之间的数据共享提供了一种较为简单的机制。当然，相关进程可能需要使用同步技术来避免彼此之间的干扰。另外，如果“共享文件”的真实目的是在两个或多个进程之间提供一个共享内存区域，那么相应的共享文件实际上并不需要作为一个文件而驻留在文件系统上。

举例来说，下面是关于如何在 Windows 操作系统的 Win32 库例程支持下创建内存映

射对象（包括文件）的说明：

```
HANDLE WINAPI CreateFileMapping(
    _in HANDLE hFile,
    _in_opt LPSECURITY_ATTRIBUTES lpAttributes,
    _in DWORD flProtect,
    _in DWORD dwMaximumSizeHigh,
    _in DWORD dwMaximumSizeLow,
    _in_opt LPCTSTR lpName
);
```

相应参数的具体含义如下：

- hFile——一个指向用来创建映射对象的文件的句柄。如果 hFile 为 -1，那么对应调用还必须通过参数 dwMaximumSizeHigh 和 dwMaximumSizeLow 来指定有关映射对象的大小，同时在系统页面文件中创建一个临时文件，而不是建立到文件系统的一个文件的映射。
- lpAttributes——一个指向相应映射对象的包含有访问控制表（Access Control List, ACL）和其他安全信息的安全描述符结构的指针。
- flProtect——用来保护相应映射对象，可具体取值为如下常量：
 - PAGE_READONLY（以只读方式打开映射对象）
 - PAGE_READWRITE（以可读、可写方式打开映射对象）
 - PAGE_WRITECOPY（以写时复制方式打开映射对象）
 - PAGE_EXECUTE_READ（以可执行、可读方式打开映射对象）
 - PAGE_EXECUTE_READWRITE（以可执行、可读、可写方式打开映射对象）
 - PAGE_EXECUTE_WRITECOPY（以可执行、写时复制方式打开映射对象）
 - 等等。
- dwMaximumSizeHigh——用来指定相应映射对象的最大长度的高位双字，即高 32 位。
- dwMaximumSizeLow——用来指定相应映射对象的最大长度的低位双字，即低 32 位。
- lpName——用来指定欲映射的文件名称。

11.6.3 Windows XP 预提取文件

为了优化请求分页的使用，各种操作系统均已开发了一些很有意思的技术。其中，在 Windows XP 中所使用的一种有趣的技术被设计成是用来提高程序加载和初始化的速度。其基本想法是，当一个程序被加载时，每次总会执行相同的指令序列。因此，这一过程总会产生相同的页面故障序列即缺页序列。同时，这项技术倾向于成群地产生这些页面故障。例如，当有关代码执行时，该项技术将会触发提取相应代码的连续页面序列。鉴于其间会牵涉到其他页面中的子程序的调用和数据的引用，所以在这一过程中还会产生其他的页面故障。最终，有关磁盘驱动器就获得了以随机顺序来回查找和提取相关页面的训练机会。XP（以及某些其他的操作系统）采用了一种更好的技术。当一道程序第一次启动的时候，有关操作系统将会监测记录该程序在前几分钟内所产生的所有页面故障，并将这些页面故障记录在称为**预提取**（prefetch）文件类型的一个文件中。接下来，有关操作系统将会在后台对该文件进行排序处理。这样，当以后对应程序启动的时候，操作系统便可以提取那些在程序初始化时所使用的所有代码页。系统可以通过几个规模比较大的读取操作来提取对应主程序所需要的所有页面。然后，系统将会移动到磁盘上的另一个地方来提取所有的子程序代码，其后再移动

到另外的一个地方来提取将要使用的那些数据页，等等。这种技术将会减少大量的缺页中断次数。同时，由于利用一次操作方式来完成较大磁盘存储块的读取，所以这种技术还极大地减少了磁头移动和旋转延迟等相关开销。

11.6.4 Symbian 内存管理

Symbian 操作系统是为在手机（cell phone，也称为移动电话）平台上使用而创建的。这种操作系统以一种独特的方式来利用现代处理器中的有关分页硬件。其间所面临的问题是：在手机中，往往假定没有辅助存储器，也就是说没有磁盘驱动器。正如第 4 章中针对 Palm 操作系统所讨论的那样，存储在手机中的所有程序总是存放在主存中。因此，与大多数系统相比，其主存更加稀缺，特别是考虑到在手机上需要保持低功耗规格的情况下。另一方面，在手机上所使用的处理器架构包含有分页硬件，而大多数的系统环境包含有辅助存储器。同时，在大多数的操作系统中，内存管理硬件应该完成的三项功能包括：1）程序的动态重定位；2）对给定程序的保留空间的寻址的限制；3）允许从辅助存储器随机动态加载任何页面到主存储器中。在 Symbian 操作系统中，并不需要动态加载功能。此外，为每个程序存储页表将会占用宝贵的内存。因此，Symbian 开发人员面临的问题就是如何最有效地利用硬件来完成其余的两项工作。Symbian 操作系统开发人员所采用的解决方案是，为系统中的所有进程使用唯一的一张页表。

当需要上下文改变并且某一程序即将进入运行状态时，该单一页表就需要进行修改。图 11-14 举例说明了相关修改的基本做法。在图 11-14a 中，我们可以看到当进程 B 正在运行时的页表。该页表拥有从进程 A、进程 B 以及它们各自的线程数据页到相应的页框的普通的映射。同时，该页表还预留有一节，始终用来指向当前正在执行的进程的页框。在应该进行上下文切换和开始执行进程 A 的时候，操作系统将把进程 A 的相关页表项复制到为当前运行进程所预留的那部分页表项中，具体在图 11-14b 中进行了例示。其中，我们看到页表已经更新为运行进程 A 时的情形。这种做法的结果就是，当前运行进程的页框的指针总是会出现在页表中的两个位置，一处是其实际驻留的地方，另一处是作为正在运行进程所对应的相关部分。这便允许有关分页硬件能够支持简化代码生成所需的动态重定位功能，同时无须为每个进程配备一个页表而消耗额外内存的情况下限制对应程序只能访问其自身的内存区域。

[251]

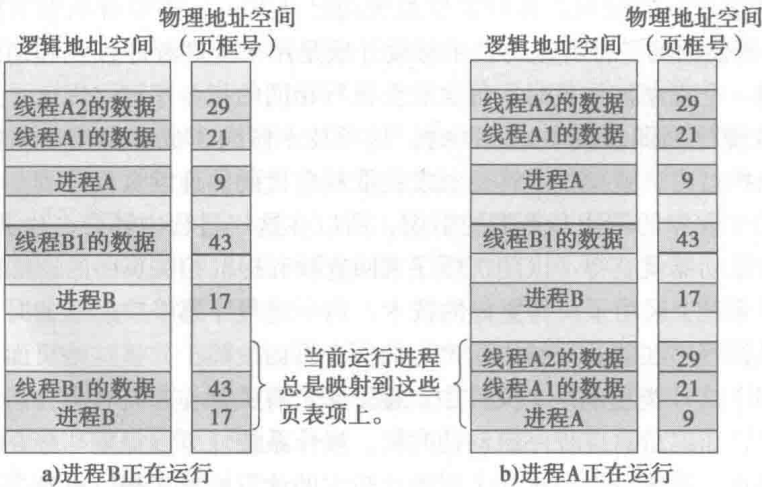


图 11-14 Symbian 内存页表

11.7 小结

在本章中，我们通过分页和分段系统及其硬件要求，还有分段和分页的结合等方面讨论了内存管理的相关设计。接下来，我们讨论了请求分页内存管理。我们分析了请求分页的效果以及在其实现中所出现的一些问题。在这个讨论过程中，我们还着重介绍了有关操作系统技术所需的硬件支持。另外，我们还用一节的内容围绕高级内存管理相关的一些主题进行了阐述。

参考文献

- Belady, L. A., "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems Journal*, Vol. 5, No. 2, 1966, pp. 78–101.
- Belady, L. A., and C. J. Kuehner, "Dynamic Space Sharing in Computer Systems," *Communications of the ACM*, Vol. 12, No. 5, May 1969, pp. 282–288.
- Carr, R. W., and J. L. Hennessy, "WSClock—A Simple and Effective Algorithm for Virtual Memory Management," *Proceedings of the Eighth Symposium on Operating Systems Principles*, Vol. 15, No. 5, December 1981, pp. 87–95.
- Prieve, B. G., and R. S. Fabry, "VMIN—An Optimal Variable Space Page Replacement Algorithm," *Communications of the ACM*, Vol. 19, No. 5, May 1976, pp. 295–297.
- Stephenson, C. J., "Fast Fits: New Methods for Dynamic Storage Allocation," *Proceedings of*
- Denning, P. J., "The Working Set Model for Program Behavior," *Communications of the ACM*, Vol. 11, No. 5, May 1968, pp. 323–333.
- Denning, P. J., "Virtual Memory," *ACM Computing Surveys*, Vol. 2, No. 3, September 1970, pp. 153–189.
- Denning, P. J., "Working Sets Past and Present," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 1, January 1980, pp. 64–84.
- Mattson, R. L., J. Geeksie, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, Vol. 9, No. 2, 1970, pp. 78–117.
- the Ninth Symposium on Operating Systems Principles, ACM, Vol. 17, No. 5, October 1983, pp. 30–32.

本章的参考文献与前一章的参考文献有着较多的重复。

252

网上资源

<http://www.symbian.com> (Symbian操作系统)

习题

- 11.1 什么样的硬件开发解决了外部碎片的问题？
- 11.2 当有关分页硬件将逻辑页面地址（即页号）转换为物理页框地址（即页框号）时，相应地址中的偏移地址部分会发生什么？
- 11.3 当我们第一次就通过在内存中的一张表格实现内存引用的地址转换方案进行评判时，其中对有效的内存访问时间的影响是什么？而我们就此采取了什么样的措施？
- 11.4 使用页表，我们需要通过某种方法来确定相应的逻辑地址空间在表中结束的位置。我们讨论了用来实现这一目标的两种不同的技术。这两种技术使用了什么机制？在什么样的情况下，其中的一种技术会优于另一种技术？
- 11.5 最终，页表开始变得非常庞大和稀疏。什么技术被用来解决这个问题呢？
- 11.6 分页之外的另一种方法是分段。请简要阐述这种技术。
- 11.7 请求分页背后的基本理念是什么呢？
- 11.8 当运行请求分页机制时，操作系统如何知道一个进程需要某个页面？
- 11.9 什么是进程的“工作集”？
- 11.10 为什么我们如此关心页面淘汰算法？
- 11.11 当发生缺页时，为什么我们不太愿意淘汰脏页？
- 11.12 对于请求分页方式而言，当操作系统选择要淘汰的页面时，局部置换和全局置换之间的区别是什么？

- 11.13 如何确定一个进程运行所需的最小页面数量?
- 11.14 如果一个进程频繁地发生抖动,那么该进程的设计人员应该如何改善这种情况呢?
- 11.15 操作系统可以执行什么样的后台操作来提高请求调页的性能?
- 11.16 就请求分页而言,散列表是非常差的执行结构。但是我们也曾提到,二分搜索却可能相当不错。还有哪些其他基本的系统结构类型会提供非常好的请求分页性能呢?
- 11.17 Windows XP 中设立预提取文件类型的目的是什么?
- 11.18 多个进程是如何使用内存映射文件的?
- 11.19 Symbian 操作系统以一种非常特殊的方式来使用分页存储器硬件。这是为什么呢?

面向深度的操作系统概念的展示： 文件系统和输入 / 输出

并不是所有的操作系统都有文件系统，但是，通常被我们视为计算机的任何设备肯定会有一套文件系统。事实上，许多不被我们看作计算机的设备，包括许多游戏系统、移动电话、音乐播放器和掌上电脑等，也可能有文件系统。这一部分将涵盖操作系统中与辅助存储器管理及辅助存储器上所见到的文件系统相关的一些内容。

第 12 章讨论了典型硬盘驱动器的布局，说明了文件系统所拥有的基本功能。该内容从目录概念及其在现代文件系统中的应用方案出发，讨论了文件存取方法（包括顺序存取、随机存取和索引存取）的基本原理，介绍了文件系统空闲空间的监测以及文件自身的组织（或分配）。

第 13 章首先介绍了几种现代的文件系统，将它们作为案例研究，具体说明了第 12 章中所讨论的各种机制是如何运用到实际的操作系统中的。该内容涵盖那些对于通常的应用不那么基础但却在现代操作系统中经常见到的高级的文件系统功能。相关主题包括虚拟文件系统 / 重定向、内存映射式文件、文件系统实用例程以及基于日志的文件系统。

第 14 章则把视野转向了常常独立于文件系统的较低层级，讨论了在各种操作系统中所呈现的完整的输入 / 输出管理子系统，介绍了各式各样的输入 / 输出设备，包括那些被用做辅助存储器的输入 / 输出设备。该内容之所以纳入这一部分，是因为辅助存储器的管理是输入 / 输出子系统的最主要的应用。输入 / 输出的其他方面则被单独分离出来，放到如关于网络连接的章节中进行说明和介绍。

文件系统基础

文件是操作系统能够提供的最重要的抽象概念之一。文件概念的出现要先于计算机，所以文件便成了人人都懂的说辞。程序设计人员并不想在磁盘驱动器、磁带或者其他任何媒介上多花费心思，他们绞尽脑汁的是他们正在处理的数据，而且他们把数据看作是一个集合。在一台计算机中，数据的集合被抽象为一个文件。程序需要针对数据进行加工处理。我们常常把数据存放在辅助存储设备上，因为主存储器成本太昂贵了，所以不能用来保存我们需要访问的所有数据。现在，这些设备大多数是旋转式的磁盘驱动器。作为应用程序设计人员，我们不想纠结在成千上万种不同类型的磁盘驱动器的操作细节方面，我们更愿意从某种抽象层次来看待我们的数据。通常，我们认为一个文件就是由若干记录或者字节组成的一个集合。因此，大多数操作系统的一项主要功能就是提供辅助存储器上的文件的抽象机制。一个文件的内容通常只对应用程序来说才是有意义的，由此我们想要说明的是，操作系统通常并不清楚文件的内部结构。但有少数情况例外，比如操作系统可以运行的可执行（二进制）程序以及被用来生成那些文件的目标模块。相关文件拥有由操作系统自身所定义的结构，且有关结构应当被所有用来生成可执行文件的链接程序（linker）或装入程序（loader，或称为加载程序）等实用工具以及用于把源程序文件转变生成目标模块的编译器（compiler）和汇编器（assembler）所掌握。

在第 12.1 节，我们将介绍文件系统的概念以及文件系统是如何装配到操作系统中的。现代的计算机通常包含几十万个文件，必须能够有效地组织文件以保证可以找到有关的数据。为此，我们紧接着讨论了用于支持文件系统的目录的相关机制。关于如何访问文件中的数据，不同的应用程序有不同的要求。有时，数据可以是按顺序连续地进行处理。而有些时候，处理则是随机的。还有一些时候，通过一个特定的关键字，很容易就能找到一条记录。而其他时候，我们则需要根据内容来访问相应的记录。在第 12.3 节中，将会阐述应用程序能够用来访问文件中的数据的各种各样的方法。随机存取介质上的文件系统需要记录该介质的哪些部分包含有数据，哪些部分是空闲和可以使用的。因此，接下来我们将会探究文件系统中当前尚未分配给任何文件的空闲空间的监测需求以及用于跟踪空闲空间的不同结构。在第 12.5 节中，我们探讨了关于文件本身结构的话题，讨论了各种方法的权衡问题。最后，在第 12.6 节中，我们对全章内容进行了总结。

[257]

12.1 引言

文件系统通常按分层结构进行设计，且每一层都为它的上层提供服务。关于这些层次的功能划分，各类操作系统都有一套独特的划分方案。不过，有两项划分原则对于所有的操作系统都是成立的：一是顶层的应用程序接口（API）应建立起文件概念的抽象机制，二是底层直接和硬件进行交互。举例来说，Linux 文件系统组织如图 12-1 所示（图中各层从左到右列出，分别对应自顶向下的各层）。我们在本章主要讨论文件抽象机制，关于底层部分将放到下一章进行讨论。

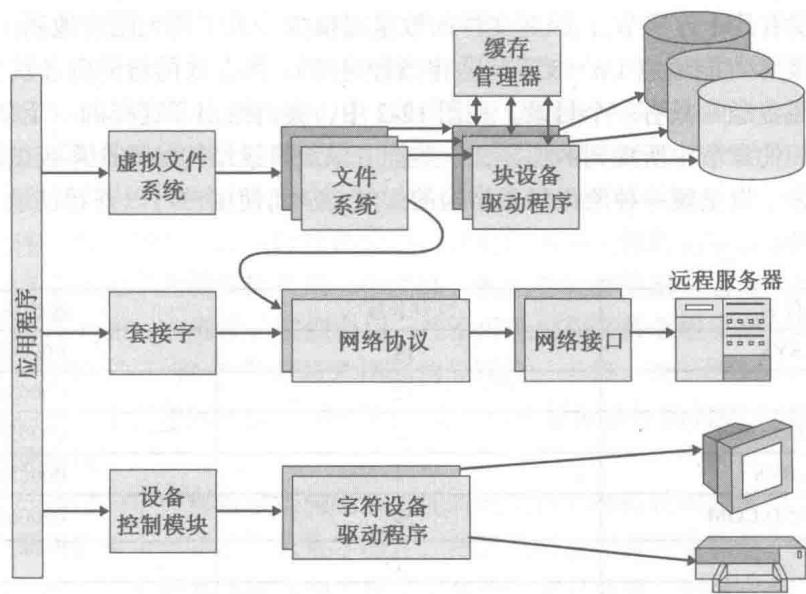


图 12-1 Linux 文件和输入 / 输出系统

258

12.2 目录

在程序使用文件之前，其首先需要找到该文件。操作系统需要为程序要搜索的文件提供某种索引，我们称这些索引为目录（directory）。（随着越来越多的对计算机并不非常了解的人也开始使用计算机，关于这些结构的另一个术语，即文件夹（folder），也同时得到了大家的普遍使用。）毫无疑问，目录会用来存放文件的名字，但是目录也会同时存放关于文件的其他数据。在一些操作系统中，对于每个文件可能会有很多其他的数据需要保存，而在另外一些操作系统中，则仅需保存关于文件的很少量的其他数据。这里，关于文件的其他信息并不是文件数据的组成部分，称之为文件元数据（file metadata）。其中的某些信息项几乎是通用的，而另外一些则非常少见。显然，我们需要一个磁盘地址来指向文件数据的开始位置。而一般情况下，我们也想知道文件的大小。表 12-1 中列出了我们在各类操作系统中可能找到的关于文件的一些元数据的例子。任何操作系统都不可能拥有所有这些信息项——在某些情况下，某些信息项甚至代表实现相同目的所采用的不同方法。在一些操作系统中，相关信息存放在文件的目录项中。而在另外一些操作系统中，则被存放在一个独立的结构中——最值得一提的是，由 UNIX 衍生出来的各类操作系统一般采用所谓索引结点（inode）的一种外部表里来存放相关信息。

12.2.1 逻辑结构

有许多不同的逻辑结构可用来存放文件系统的目录结构。在本节中，我们将介绍几种常用的结构。

单级目录结构

我们如何在逻辑上组织一个磁盘的目录，在某种程度上取决于磁盘的大小。就像我们在第 4 章中所讨论的那样，早期的磁盘

表 12-1 一些可能的目录信息项

文件名	是否归档?
→起始块号	保护类属性 (可能非常复杂)
最大文件长度	加密信息
当前文件大小	压缩信息
→结束块号	所有者标识符
创建日期和时间	文件分配类型
最后被修改的日期和时间	最后被访问的日期和时间

空间相当小（仅有几十万字节），因而文件的数量也很少。为了最大限度地利用有限的空间，存放的文件名要尽可能地短（6~8 字节是相当普遍的），而存放的指向磁盘块的指针也要尽可能小。整个磁盘通常只有一个目录。在图 12-2 中，我们给出了这样的一个单级目录结构。

正如我们在前面的章节中所提到的那样，一些拥有单级目录结构的操作系统试图通过把组名与文件关联起来，以呈现一种两级目录结构的假象，从而使用户可以查看目录并且只能看到特定组别的文件。

259

文件名	文件长度	起始块号
MSDOS.SYS	14	0000404
IO.SYS	12	0000303
AUTOEXEC.BAT	2	0000505
CONFIG.SYS	1	0000506
COMMAND.COM	50	0000600

图 12-2 单级目录示例

树形目录结构

伴随时间的推移，磁盘存储容量已经有了显著的增长。对于当前的磁盘驱动器技术而言，拥有容量为数千亿字节（即几百 GB）的硬盘是一台典型的新型个人计算机的标准配置。对于这样的磁盘，上面承载有数十万个文件是很正常的。一般用户对于其中的很多文件根本不了解或毫不知情。面对如此庞大的硬盘，单一目录是无法发挥作用的。为此，在磁盘目录的逻辑结构组织中，做出的一个关键的改进就是支持多个目录。其主要的技巧就是，除了允许目录指向文件之外，还允许目录指向其他的目录。如果我们限制此类链接引用仅指向那些不会拥有其他链接而指向自身（包括起始目录）的目录，则由此产生的结构就是一棵以起始目录作为树根的树形目录结构。举例说明如图 12-3 所示。

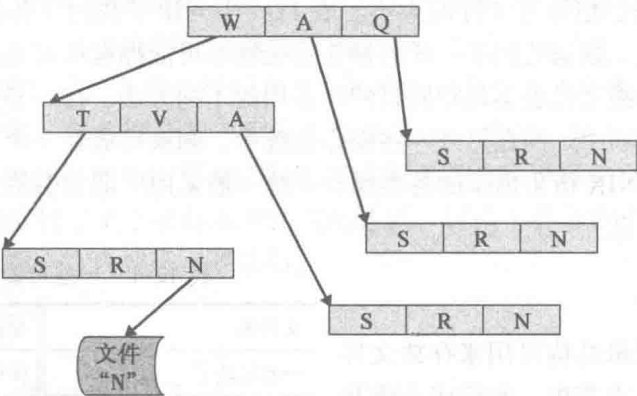


图 12-3 树形目录结构示例

有了这样的层次式目录组织，我们就可以将文件划分成不同的类别。在两个以上用户使用的机器中，我们可以给每个用户设立一个主目录，其中包含有分布在各子目录下的相应用户的所有数据文件。各式各样的子目录也可以用来组织不同类型的文件——其中有的可能用于实用程序，有的用于游戏软件，有的用于电子邮件，等等。目录划分的这一过程可以延伸

260

到任意深度。例如，电子邮件目录可以按与工作、学校、家庭、朋友和技术等相关而进一步划分为若干目录，而学校目录还可以按每个班进一步划分为更细一级的目录。

这种目录组织方式会带来一种副作用，即我们可以拥有许多相同文件名的文件，只是需要把它们存放在不同的目录中。这将允许一组在不同主目录中进行操作的用户使用相同的文件名。图 12-3 给出了一个看似不太可能但却完全合法的例子，其间许多子目录中包含有相同文件名的文件分组。但是，这一特征是要付出代价的——文件的名称就不再可能由单个的名字而唯一指定了。为了无歧义地引用一个文件，我们必须给出由通向相应文件的目录所组成的完整的路径（path）。通常，我们会用一些不可能用作文件名组成部分的分隔符把子目录名分隔开来。字符“/”和“\”是最常用到的分隔符。于是，对于图 12-3，为了无歧义地确定图中所显示的一个文件如“N”，我们将不得不给出类似这样的名字“\W\T\N”。

无环图目录结构

遗憾的是，通过一个树形目录结构是不可能对现实世界进行精确建模的。例如，金丝雀是一种鸟。如果我们有金丝雀的一张数字图片，并且我们正在研究生物学，那么我们有可能把这张数字图片放在拥有猫和其他动物的相关信息的一个目录里。同时，金丝雀还会飞，为此，如果我们正在研究工程学，我们也可能把这张数字图片放在拥有飞机和其他会飞的东西的相关信息的一个目录里。还有，金丝雀是黄颜色的，所以，如果我们是艺术家，我们还可能会把这张数字图片放在拥有黄油、柠檬及其他黄色物品相关信息的一个目录里。但是，如果我们正在研究生物医学工程和颜色视觉系统，我们就可能对这个文件如何分类无所适从和不知所措了。如果只有树形结构的目录，我们常常会为该如何对一些文件进行分类而左右为难。随之而来的是，以后或许我们不能记起当初把金丝雀的图片放到了哪个文件夹。有时用来摆脱这种困境的一种解决方案是允许目录构成有向无环图（Directed Acyclic Graph, DAG）。为了实现这种方案，会用到一种所谓别名（alias）的特殊类型的目录项。别名目录项并不直接指向一个文件，而是指向另外的一个目录项。（事实上，别名目录项可以指向文件，但是那样会引发一些问题，具体我们放到以后进行讨论。因为，这两种实现机制之间的差别与这里的内容没有多大关系。）

遗憾的是，从树形结构转变为有向无环图将带来一些问题，必须要予以考虑和处理。最简单的例证是，当有程序试图去统计系统中所有文件占用的所有空间的时候所产生的问题。如果不考虑别名目录项，那么对于有一些文件不止一次被引用的情形，相应程序就可能得出错误的空间总量结果。另外一个大的问题是，系统应该如何裁决一个文件可以真正被删除掉。考虑如图 12-4 所示的情形。其中，我们可以看到三个目录。顶层目录有两个目录项分别指向了子目录 W 和 Q，同时还有一个目录项指向了文件 A。而且，子目录 W 中也包含有一个目录项指向了文件 A。假设用户从子目录 W 中要删除文件 A，但因为在顶层目录中还存在对该文件的其他引用，所以操作系统实际上并不应当移除该文件。

261

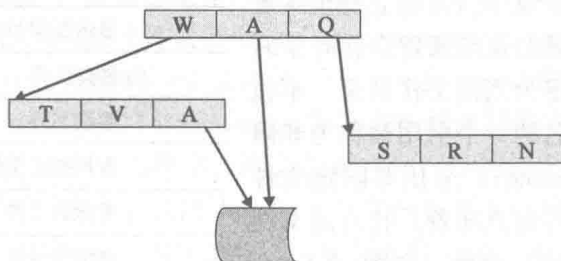


图 12-4 两个目录项指向一个文件的情形

有两种机制有时被用来解决这个问题。第一种技术就是区别对待文件的主引用和文件的任何别名。操作系统在主目录项中还应包含一个引用计数。当为一个文件增加了一个别名时，相应的引用计数应当递增。而如果删除一个别名，对应文件的引用计数应当递减，且如果计数变为零，对应文件就可以被真正删除掉。不过，这种机制在主引用已被删除而别名依然保留的情况下，会存在一些问题。第二种技术则是把所有的别名变成符号式引用 (symbolic reference)，使其包含相应的路径信息。早些时候我们所说的别名应该指向对应文件的目录项，而不是文件本身，就是这个意思。在这种情况下，如果图 12-4 中的下层的引用是主引用，那么顶层目录的第二个目录项实际上应当包含的是“\W\A”，而不是一个指向磁盘上对应文件的指针。

12.2.2 物理结构

早期的系统把相当多的注意力都放在了目录检索速度方面。为此，早期的系统有时会使用诸如散列之类的技术来提高目录检索速度。然而，在最近 20 年左右的时间里，处理器和内存提高的速度比磁盘驱动器提高的速度至少要快 10 个数量级以上。因此，大多数现代的操作系统并不担心此类问题，而且目录也不是按照某种特定顺序进行过排序的。对目录的检索只是按顺序方式进行。大多数情况下，人们倾向于把目录维持在比较小的水平上——大致在 100 个目录项以下。

12.2.3 目录操作

操作系统必须支持关于目录的各种不同类型的操作。有人可能会认为这些操作只是那些用来支持文件的操作，因为目录本质上就是文件。但是，文件和目录之间还是存在一些区别的。首先，考虑到文件系统包含的一点错误就可能引发潜在的灾难性后果，所以大多数操作系统并不允许应用程序直接对目录进行写操作。相反，应用程序必须通过调用特殊的例程才能创建一个新的文件或目录，或者完成关于目录的任何其他操作。表 12-2 给出了操作系统可能支持的一些目录操作类型。

262

表中列出的第一项操作是改变工作目录。就像前面所提到的，每个子目录中可以包含与其他子目录中相同本地名称 (译者注：实际上就是文件名) 的文件，因此，需要一个路径名来无歧义和明确地指定一个文件。当我们在命令行中输入文件名和运行一个程序的时候，我们并不想总是无可奈何地输入路径名 (译者注：路径名经常比较长)，所以操作系统便采用了一个所谓工作目录的概念。基本想法是，用户应当采取一些措施来指定一个特定的目录作为当前工作目录 (current working directory)，有时干脆就称作工作目录。一种可以确定工作目录的方法是记录相关信息到系统中，并建立健全系统登录环节。支持相关登录的系统通常会在登录时对应用户的主目录指定为当前工作目录。不包括任何路径信息的文件名的一个引用被称为非限定文件名 (unqualified name)。采用非限定文件名方式引用的任何命令都将意味着，相应的文件应该位于当前工作目录中。为此，在图 12-5 中，

表 12-2 操作系统应当支持的目录操作类型

改变工作目录
创建目录
删除目录
显示指定目录的文件列表
创建文件
删除文件
查找指定文件
重命名文件
遍历目录树

如果目录 W 是当前工作目录，那么对文件 S 的引用就会被认为是对该目录下的对应文件 S 的引用。而为了引用目录 W 下的子目录 T 中的文件 S，程序则必须要把通往该目录的路径作为指定名称的组成部分。这种情况下，可以为“\W\T\S”或者“.\T\S”。第一种引用方式采用的是**绝对路径名**（absolute pathname），其以用来分隔路径名中各目录名的分隔符打头，故而可以被解析为起始于目录树的根部（译者注：即所谓的根目录）。第二种引用方式采用的是所谓的**相对路径名**（relative pathname），此处的“.”是用来指定当前工作目录的一个特殊的名字，所以该路径名表示的是，对应路径从当前工作目录开始，然后到达子目录 T，进而在那里就会找到文件 S。

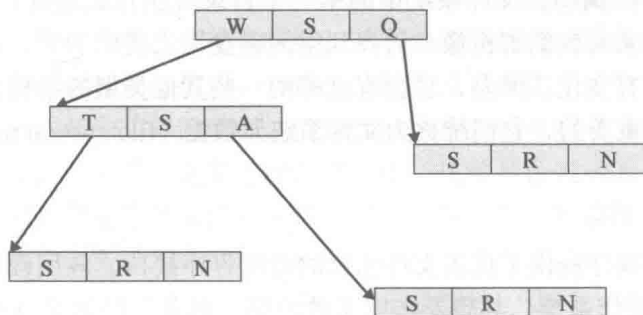


图 12-5 目录结构中的路径

263

另外一种改变当前工作目录的常用机制是使用改变目录（change directory）的命令——通常是像“cd”或“chdir”之类的命令。该命令可以指定绝对路径，也可以指定相对路径。通常，可以采用简化符号，例如，“cd..”将会把当前目录的父目录变为工作目录。在 Linux 和其他类 UNIX 系统中，不带任何参数的“cd”命令将会把用户的主目录设立为当前工作目录。

关于目录创建和删除的命令非常简单。再次强调，这些功能函数之所以存在，是因为我们通常不允许应用程序直接对目录进行写操作。相反，我们要求它们通过使用特殊的操作系统调用来完成这些操作功能。同时，还有一些特殊的实用例程 mkdir（译者注：用于创建目录）和 rmdir（译者注：用于删除目录）支持用户通过命令解释器来请求相应操作。通常情况下，操作系统可能不会提供内置功能函数来列出普通文件的内容。但是，因为目录是一类非常特殊的文件，所以操作系统必须要提供一个功能函数来为应用程序列出目录的内容。同样，也有相应的实用例程（dir 或 ls，译者注：用于列出指定目录下的文件）用来支持用户通过命令行接口来访问对应的功能。然而，当应用程序需要创建一个新的文件时，其必须拥有一种途径来要求操作系统去执行相应的操作。类似地，程序可能想要删除一个不再需要的文件。一般情况下，不会专门设立此类简单的实用例程来创建一个新的文件，因为这样的文件是空的，意义不大。为此，文件往往是作为某些其他操作的副产品而被创建的。关于此类操作，最接近或具有实用性质的是文件复制命令（cp 或 copy）。在 Linux 平台上，用户可以把特殊的伪文件“/dev/zero”复制到一个文件名，从而创建一个二进制的零的流文件。当然，文件常常通过使用像 vi 或记事本（notepad）之类的文本编辑器等实用程序来进行创建。还有一些如在微软办公软件中的应用程序，则会创建它们自己的特定类型的文件，如 .doc 文件或 .xls 文件。文件删除则通常会以像 del 或 rm 之类的具有文件删除功能的实用例程提供给用户。删除目录也是一种特殊的实用例程，一般拥有像 rmdir 之类的名字。另外，搜索目

录来查找文件也常常是应用程序需要完成的事情，其目的并不是打开文件用于输入。操作系统（或编程语言库模块）一般会提供此类功能。确切地说，当应用程序想要创建一个新的文件时，其需要进行目录搜索和文件查找。这种情况下，首先应当检查确认在当前目录中没有使用相应指定的文件名。（某些编程语言库也可能实现这样的功能。）

12.2.4 文件系统元数据

我们之前提到过，目录项所包含的关于文件的信息并不是文件本身的组成部分，相关信息称作文件元数据。在文件系统中，还有一些其他的信息不是关于特定文件的，所以它们不是目录项的组成部分。例如，文件系统中的一个目录放在什么地方？后面我们将会看到，有一些其他的结构用来向我们解析像如何找到空闲磁盘块之类的问题，具体实现细节对于特定的文件系统往往会有变化。但是，总会有这样的一些其他类型的结构，并且它们对于文件系统的完整性是非常重要的，它们统称为文件系统元数据（file system metadata）。

264

12.3 存取方法

操作系统为应用程序提供了代表文件抽象的应用程序接口。应用程序接口应当包括关于“应用程序如何告知操作系统，其想要访问文件的哪一部分”的语义。不同的应用程序需要不同的存取模式及存取方法（access method，或称为访问方法）。

12.3.1 顺序存取

最初，计算机应用程序被设计成按批量方式来进行信息的处理，且相关信息已经根据某种关键信息（譬如零件编号或客户编号等）进行了排序。这样的应用程序需要顺序地处理文件。曾经有一段时间，相关文件是按照字面排好序的穿孔卡片叠，后来才变成了存放在磁带上的排好序的数据块。系统可能拥有一个关于事务处理的诸如计时卡片之类的输入文件和一个诸如工资记录的主文件，且它们可能都按员工编号进行了排序。应用程序将从每个文件的开始部分启动文件读取操作，并且一步步渐进式地对每个文件进行读取，按关键字段（此例中即员工编号）保持它们的同步。对于卡片叠来说，记录的大小是固定的。而对于磁带来说，记录可以是任意一个合适的大小，一直到由硬件或操作系统所决定的某个最大值。对于磁盘存储器的顺序处理来讲，操作系统（或软件库）必须拥有关于各文件的记录大小的某种定义，从而使其能够为持有打开文件的每个应用程序记录当前位置（或当前记录指针），如图 12-6 所示。（注意，访问相同文件的不同进程有可能会有不同的当前记录指针。）对于标准的顺序处理（或称为顺序存取或顺序访问）的每次读操作或写操作而言，操作系统将会为之对当前记录指针执行递增处理。同时，应用程序接口中还常常会设立一条命令用来重置当前记录指针，使其指向文件的开始位置。这项操作有点类似于将磁带倒回到起始位置的操作类型。鉴于磁盘块是固定大小的，而且不可能刚好匹配上应用程序的记录长度要求。所以，对于操作系统来说，将两条以上的逻辑记录组合起来存放到一个

265

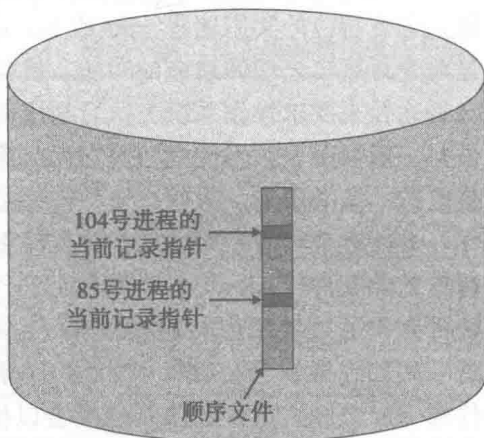


图 12-6 顺序文件及当前记录指针

个物理数据块中是相当常见的。在下一章中，我们将会更全面地介绍分块的有关内容。

12.3.2 随机存取

因为磁盘驱动器越来越便宜了，所以辅助存储器机制就从存储在磁带上变成了存储在磁盘驱动器上。一旦数据大部分是在线保存的，就有可能在各项事务发生时对其进行及时地处理，而不是把它们积攒下来，然后再去顺序地批量处理。一般而言，事务处理比批量处理更为可取，是在于其支持管理人员更近乎于实时地去记录和跟踪企业的运行状况。然而，这意味着应用程序必须要随机地，而不是纯粹顺序地访问主文件数据。为此，需要对文件的应用程序接口进行扩展，以涵盖另一种模型，即随机存取（random access，或称为随机访问）模型。在该模型中，应用程序将告知操作系统，其需要访问文件的哪条记录，而操作系统将会直接移动到对应记录，对其进行访问，并执行读操作或写操作。通常情况下，这将需要从关键字取值到记录编号的某种简单的映射。例如，一家小公司可能只是顺序地分配了员工编号，并把员工编号作为记录编号。在某些操作系统中，这种寻址表示为一个记录编号，而在其他的操作系统中，则被表示为从文件的起始位置开始算起的字节偏移量。

注意，随机存取文件也可以实施顺序存取方式。当应用程序随机地访问一条记录时，其将会抛弃和暂时离开用于定位到下一条记录的当前记录指针。这时，应用程序还可以发出一条 read next（即读取下一条记录）的操作命令，于是操作系统将会返回下一条记录，并将当前记录指针递增。在图 12-7 中，我们可以看到这一场景，其中，34 号员工的员工编号用来访问文件中的对应记录。如果应用程序执行了 read next 操作，其将获得当前记录的下一条记录。

266

为了能够从随机存取文件的任何位置都可以开始访问，操作系统常常会提供一条 seek 命令（译者注：关于文件读写指针随机定位的命令），用于把当前记录指针定位和指向到对应关键字取值等于或大于给定键值的第一条记录。当操作系统一次只运行一个进程时，这条命令实际上会将磁头定位到文件的相应位置（即其将会查找和定位数据的物理位置）。现在，它仅仅是一种逻辑定位而已。

12.3.3 更高级别的存取方法

大多数操作系统至少会提供对顺序存取方法和随机存取方法的支持。一少部分操作系统还会提供一种或多种更高级别的存取方法。我们将在本节的剩余部分，就两种此类的存取机制加以阐述。大多数此类的高级存取方法还包含在数据库系统中，有时也作为库模块提供对高级语言的支持。拥有操作系统提供的存取方法意味着，只要保证操作系统的存取方法对相关高级语言进行支持的应用程序接口的语义足够相近，那么仅需完成很少的开发工作就可以支持许多高级语言。

索引存取

对于大公司的员工文件的处理而言，随机存取往往不能像处理小公司的员工文件时那么有效。时不时地，可能会有许多员工退休、离开公司或者被

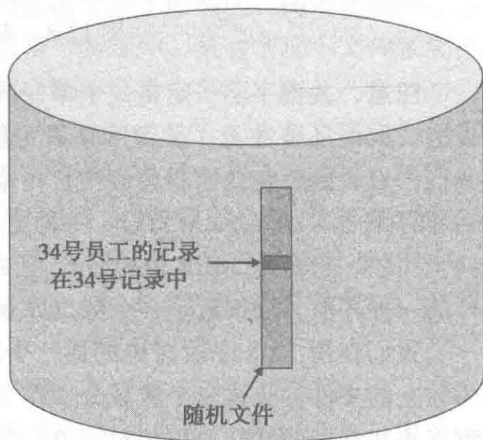


图 12-7 随机存取文件及存取方法示例

解雇，等等，其结果是在主文件中将会出现许多不是描述现有员工的记录。在这种情况下，操作系统可以提供一种称为索引存取（indexed access）的存取方法。此类存取方法的一个相当常见的术语是索引顺序存取方法（Indexed Sequential Access Method，ISAM）。

这种存取方法的工作机理如图 12-8 所示。图中显示的是关于一家零售店的数据文件。该文件包括三个区域：用于保存数据记录的主数据（primary data）区域、对记录中主关键字字段进行索引的主键（primary key）区域以及对另外一个变量（即辅助关键字）进行索引的次键（secondary key）区域。伴随记录不断地被添加到文件中，相关记录将会顺序地写入主数据区域。然而，对于每一条被写到主数据区域的记录来说，都会有一条额外的记录被写到主键区域，同时还有另一条记录被写到次键区域。（注意，可能没有次键区域，但也可能有多个次键区域。）主键区域和次键区域的各条记录均按相关联的关键字字段取值的大小有序存放。在图中，用来索引数据的关键字字段有两个：零售店库存编号和制造商商品编号。为此，当 0 号记录被写入主数据区域的时候，一条记录就会被写入主键区域（该记录用以说明零售店库存编号 ABC 在 0 号数据记录中被找到），同时还有另外一条记录被写入次键区域（该记录用以说明制造商商品编号 CBA 在 0 号数据记录中被找到）。同样，当第二条记录写进主数据区域的时候，类似的记录也会被写到主键区域和次键区域。但是，当添加第三条记录时，被添加到主键区域的那条记录会引发一个问题，原因在于若直接添加该条主键记录，将不再保证主键区域的顺序性。为此，我们不得不通过关键字的取值来对这个区域进行排序。有很多构建键码区域（译者注：即主键区域或次键区域）但回避对整个文件进行实际排序的技术，包括二叉树（binary trees，或 B 树）、散列法以及多级索引。

267

索引	零售店			零售店		制造商	
	库存编号	商品编号	其他信息	库存编号	索引编号	商品编号	索引编号
0	ABC	CBA		ABC	0	ABQ	3
1	XYZ	JKL		ABQ	3	CBA	0
2	MNO	CBA		MNO	2	CBA	2
3	ABQ	ABQ		RST	4	JKL	1
4	RST	UVW		XYZ	1	UVM	4
5		未使用部分		未使用部分		未使用部分	
6		
7							
主数据区域				主键区域		次键区域	

图 12-8 索引存取方法及相关文件示例

注意，关键字不一定是一个单一的字段。例如，可能会创建一个把姓和名联系到一起的索引。同时还要注意，关键字字段可能允许键值重复，也可能不允许键值重复。图 12-8 中我们可以看到，在这家零售店中，一个单一的制造商商品编号（CBA）拥有两个不同的零售店库存编号（ABC 和 MNO）。一种更可能的情况是，在我们的员工文件中，我们可能会有两个比尔·史密斯（Bill Smiths），但是我们不应该有两个员工拥有相同的社会保险编号。这样的一种存取方法接近于一个数据库系统，但是有点简单。

我们在图 12-8 中所讨论的那三个区域可能是单个文件的组成部分，也可能是分别存储为独立的文件。让这三个区域分别作为独立的文件存储，可能会使文件创建之后的为另一关键字添加索引的操作变得相对简单一些。让三者成为单独文件的风险在于，对文件进行备份和恢复的时候，最终很容易会得到无法合到一起的三个文件（译者注：各文件数据出现了不

同步的情况)。当然,我们很可能会拥有一个用来验证并可能重建次键索引文件的实用程序。但是,对于一个大的文件来说,这可能会非常耗时,而且我们往往不可能及时意识到当时发生了问题和我们应该运行那个实用程序。

散列存取

操作系统有时也会提供另外一种高级别的存取方法,即散列存取(或称为散列访问)方法。在键值没有全部被用完的情况下,计算关键字字段的散列值可以被用来创建一个随机键值用于快速地访问一个随机存取文件。毋庸置疑,生成散列键值可能会创建一些因为源关键字取值不同而发生冲突的记录编号,因此必须要提供一种机制来消解这些冲突。虽然散列文件存取方法不像索引顺序存取方法那样普遍,但是其也是操作系统提供的一种非常有用的工具。

12.3.4 原始存取

对于某些应用程序而言,使用文件系统提供的服务可能无法达到预期结果或效果。在应用程序对性能要求比较高且应用程序开发人员对相关文件访问模式非常了解的情况下,就可能会发生这种问题。究其原因是在于,为大多数应用程序设计的服务是面向那些对文件处理要求并非异乎寻常的所谓“普通的”或“典型的”应用程序而提供的。为解决这种问题,操作系统有时会提供一个所谓的**原始存取方法**(raw access method)。其间,操作系统不会提供任何文件结构,而是预留了一块磁盘区域用于存放应用程序所提供的自创的文件结构。此类原始存取能够派上用场的实际例子包括操作系统本身的分页存储和数据库系统。

268

12.4 空闲空间管理

操作系统把文件和目录存放在磁盘的盘块里。为了能够做到这一点,操作系统就需要记录和监测还有哪些盘块尚未被使用。通常可采用两种方法来记录磁盘的**空闲空间**(free space):链表法(linked list)和位图法(bitmap)。最初,文件系统记录的是磁盘上可被访问的最小的块空间,即扇区(sector)。伴随磁盘空间的扩充,磁盘上用于指向扇区的指针的大小也变得越来越来大。例如,现代的磁盘驱动器如今正迈向万亿字节(TB)的空间范围。任何一个超过2TB的驱动器都要求相应指针应大于4B(译者注:其实应为大于5B)。当然,起初为软盘设计的文件系统并没有用到那么大的指针。因此,当磁盘空间膨胀过大使文件系统相关指针不再适用时,一种简单的解决方案就是一次分配两个以上的扇区。简单地按两个扇区一起进行分配会使对应指针的涵盖范围翻倍。这一过程不断延伸,某些情况下文件系统甚至会扩展到一次分配多达64个扇区,尽管4KB大小的分配要更为常见些。由此产生的结构被称为一个**盘块**(block),有时也被称为一个**簇**(cluster)。这种方法看起来很不错,但相关机制存在的一个问题是,如果存放在磁盘上的数据包含有许多小文件,就会很浪费空间。例如,大部分的脚本(或批处理)文件往往仅仅只有几行文本,它们都很少能填满一个扇区,更不用说64个扇区了!不过话又说回来,既然这种一次性分配多个扇区的技术非常普遍,所以在这一章中,我们通常会说分配一个盘块,而不是分配一个扇区。

12.4.1 链表式空闲空间监测

监测空闲空间的方法之一就是把所有的空闲盘块放到一个链表中。图12-9显示了某磁盘驱动器上的盘块分布情况。操作系统必须要记录和掌控链表中的第一个盘块,而每一个空

索引块来进行空闲空间的监测管理，且内存索引块与磁盘索引盘块的内容应保持一致。此处应指的是内存索引块的内容。) 写回到磁盘，这样索引盘块内容就会与现状保持一致。不过，这里往往会采取一些细微的优化处理，即常常会同时取出若干个盘块指针，然后一次性重新写回索引盘块中，并暂时将这些盘块指针存放在内存中。这一过程就是所谓的预分配 (preallocation)。这种技术可以用于许多空闲空间记录监测机制中，以便将磁盘上数据的更新减少到最低程度。当然，存在系统失败的某种可能，也就是说，磁盘上的信息可能显示某些盘块正在被使用，而其实它们并未被使用。让系统决策失败是一种概率相当低的事件。但如果系统确实发生了判断失误，我们失去监控的那几个盘块也通常是可用空间的很小的一部分。无论是文件数据还是元数据都不会存在丢失的问题。同时，我们还会有拥有文件系统检查功能的实用例程，将会以扫描文件系统为代价来恢复那些丢失的盘块。因此，我们不用担心元数据中可能丧失一致性的问题。

270

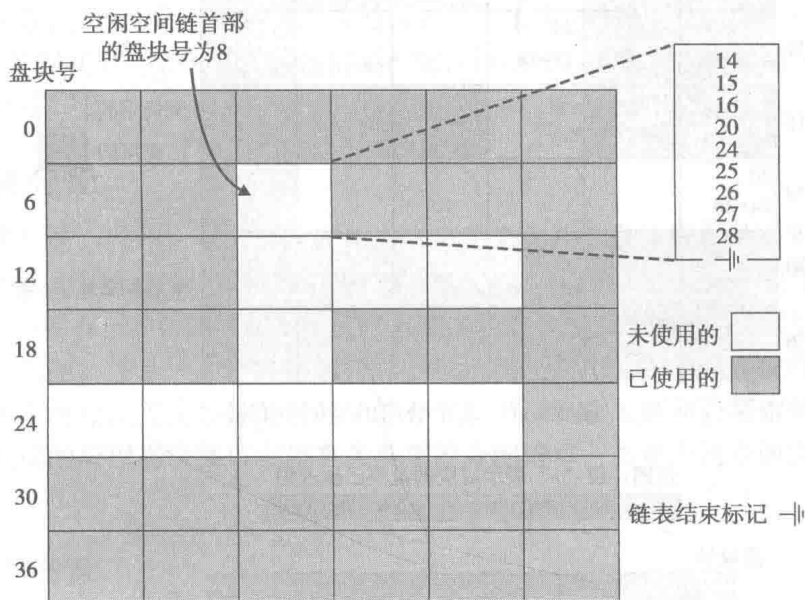


图 12-10 基于索引机制的空闲空间链

分组法是另外一种能够改进链表式空闲空间监测的机制。该项技术中，操作系统会把握每一次机会来限定链表中的两个或多个盘块是相邻的。如果每次能以多个盘块而非唯一一个盘块的方式进行文件空间的分配，那样就很容易能够做到。在这种情况下，对应组中的第一个盘块不仅仅会包含一个指向下一个空闲盘块的指针，而且还会说明链表有多少后续的盘块是彼此相邻的。相关机制示例如图 12-11 所示。这将允许分配机制在某些时候能够更容易地以连续盘块的方式进行分配。与此同时，该第一个盘块读取之后，无须再次读取磁盘，就可以把相应组中的其余盘块也分配出去。

12.4.3 位图式空闲空间监测

另外一种监测空闲空间的方法是通过一个位图，其中文件系统中的每一个盘块的使用情况都用长长的串中的唯一的一位来表征。进一步说，如果对应的位设定为这样，那么相应的盘块就是空闲的。而如果对应的位设定为那样，那么相应的盘块就是已被使用的。位“1”是指示相应的盘块为空闲的，还是指示相应的盘块已被使用，主要取决于计算机的指令集。

271

我们很快将会阐明这一点。回想一下，链表机制存在的问题之一在于其难以分配到多个连续的盘块。而借助于位图来做到这一点，将比通过链表机制要简单和容易许多。其只需要查找一串满足空间大小要求的连续的二进制位即可，具体体现为一种搜索获取连续盘块的扫描处理，并且我们希望有关扫描过程可以高效地执行。计算机的指令集可能是查找一串 0 比查找一串 1 更高效，也可能是相反的情况。这些也正是赋予了“位‘1’究竟是表示空闲盘块还是表示被占用盘块”的决定如此之重要的唯一的考量所在。位图式空闲空间监测示例如图 12-12 所示。

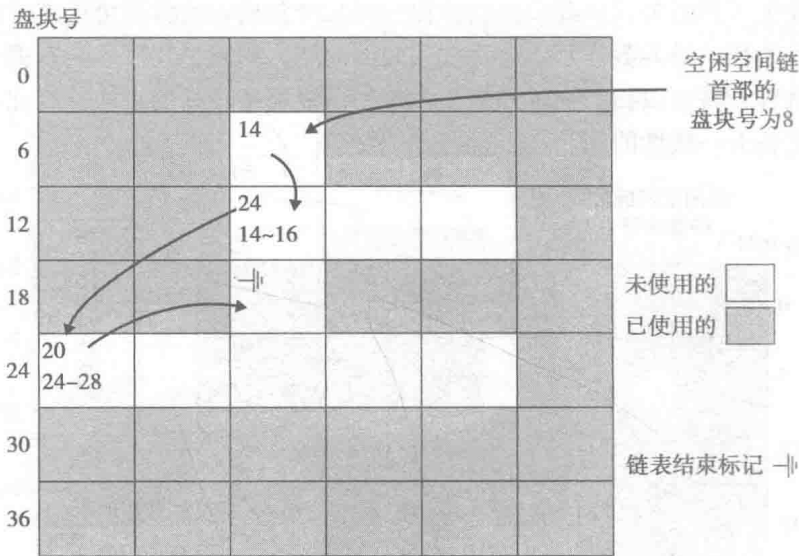


图 12-11 基于分组的空闲空间链

位图：位“1”表示对应的盘块已被占用
11111111000000111000000111100000000001100

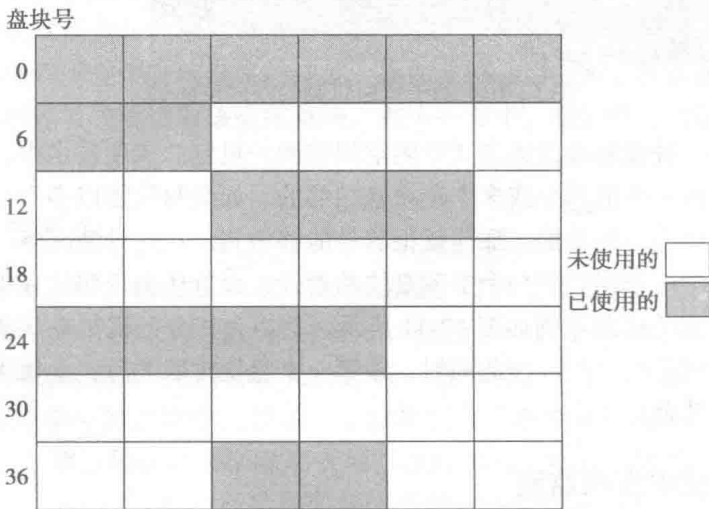


图 12-12 位图式空闲空间监测示例

注意，相对于链表机制而言，使用位图来监测记录可用的空闲空间会消耗我们更多的内存。我们需要在内存中存放位图的一部分。多数可能的情况是，我们会保存一个完整的盘块的内容，因为那样将更容易实施读取操作。而使用链表，我们仅需存放一个指针——如果我

们实施盘块预分配策略,那么可能会有更多的一些指针。尽管如此,现在的内存成本已经非常低了,而且还在不断地往下降,所以这可能并不是一项值得特别关注的考量因素。当我们分配盘块的时候,我们确实需要更新位图的磁盘副本。但是,我们也可以使用前面链表式监测机制中所讨论的预分配技术。另外,在我们真正使用盘块空间之前,需要更新映射关系,这是非常重要的。如果我们没有及时地更新,那么我们将会遇到“一个盘块分配给多个文件”的风险。那样情况就会变得很糟糕。

位图不仅会耗用更多的内存空间,也会占用更多的磁盘空间。同时,位图还必须存放在磁盘上的一个专用的地方,相应的位置不能用来进行数据存储。而在链表机制中,指针是存放在空闲盘块自身的空间里的。尽管如此,我们在这里再次强调,磁盘空间的成本要相对低廉且其价格还在继续不断地往下降,因此在现在看来,空间开销和成本可能并非主要的考虑因素,虽然其曾经确实如此。

还有一种用于监测空闲空间的更常见的机制,不过其仅仅是在文件分配表结构中用于把文件的各个盘块链接到一起的相关机制的副产物,所以我们将对应的标题下再做进一步的讨论。

12.5 文件分配

关于文件系统,另外一项主要的设计决策是关于文件自身如何在磁盘驱动器上进行组织。操作系统通过应用程序接口呈现给用户的抽象,在一定程度上决定了操作系统可以使用的组织类型。主要有三种机制可用于给文件实施空间的分配,具体包括连续分配、链接分配以及索引分配机制。注意,一种操作系统并不一定仅仅使用其中的一种文件分配机制。相反,一些操作系统往往支持多种类型的文件分配。可以确定无疑和必须做到的是,操作系统应当提供相应的应用程序接口支持有关类型的分配请求,并对空闲空间实现正确无误地监测。

12.5.1 连续分配

连续分配(contiguous allocation)意味着分配给一个文件的相应盘块的编号,共同构成一个严格地按 1 进行递增的序列。例如,在图 12-13 中,我们可以看到 File B 占据了 1000~1799 的连续盘块。这些盘块并不一定开始于磁道边界的位置,而仅仅是在编号方案中是相邻的。关于文件空间分配的这种方法具有某些方面的明显的优势。首先,只需要很少的信息就可以找到所有的数据,而所需的一切仅仅是第一个盘块的扇区地址以及以盘块数表示的文件的大小。这种分配方法会使数据的随机存取实现起来非常简单,相关确切机制会根据操作系统应用程序接口和被分配的盘块的大小而有所变化。例如,在一些操作系统中,应用程序接口常常要求应用程序向系统传入关于文件读操作开始位置的一个偏移量(用字节数来表示)以及要读取的数据的长度(通常是扇区大小的若干倍)。在这种情况下,相应的存取机制只需把字节偏移量除以盘块的大小,再将其加上文件的第一个盘块的起始扇区地址就可以展开具体的存取操作。当然,顺序存取更是小事一桩,不值一提。就像上面所提到的那样,如果采用位图式空闲空间监测机制,那么分配连续的空间毫不费力,所需完成的工作只不过是位图中查找到表征为空闲盘块的一个连续的位串即可。采用链表式空闲空间监测机制,连续分配则会很不切实际,尽管从技术上而言并非不可能,而我们所描述的分组机制或许在这一方面会有所帮助。

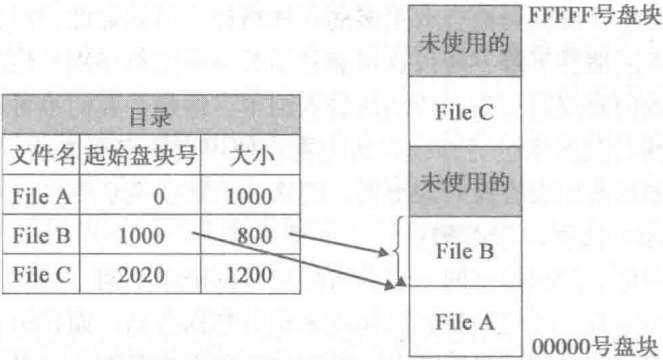


图 12-13 连续的文件分配示例

连续分配有一个问题，就是一旦一个文件被分配好了，要想让该文件变得大些就困难了，因为很有可能其他的某个文件刚好分配在我们想要变大的那个文件之后。例如，在图 12-13 中，要想不动 File B 是不可能把 File A 变大的。为了避免这个问题，编程人员往往会倾向于为文件分配比其当前数据需求要多的存储空间。那样，在文件（译者注：这里指文件空间）需要变大之前，相应文件（译者注：这里指文件数据或文件内容）是可以扩充一段时间的。例如，编程人员可能知道系统现在拥有 100 条记录，而通常情况下每个月会再增加两条记录。于是，可以使文件分配得到能够容纳 130 条记录的空间，从而保证文件可以正常运转大致超过两年的时间，并且相关空间不会被填满，同时也不需要重新分配空间。对于为文件多分配的、其暂时用不到或永远用不到而又无法分配给其他文件加以利用的空间，我们称之为程序员碎片（programmer fragmentation）。遗憾的是，这是对存储空间的浪费和挥霍。如果磁盘驱动器上有足够的空闲空间，去分配给文件的另外一个副本，那么关于文件变大的相关操作还是相当简单的。但是，如果文件很大，那么相关操作就可能是非常耗时的。如果没有足够的空间来分配给新的副本，那么对应文件就必须卸载到三级存储设备上，旧的文件被删除掉，其他的文件被搬来搬去，以便为相应的新文件腾出足够大的连续的空间来，然后再为该新文件分配空间，最后将数据加载到新文件里。

这一过程的笨拙引发了连续分配机制的一个变革——对扩展盘块区的使用。在该方案中，一个文件不再局限于唯一一次的连续分配。初始分配得到的是一个连续的盘块区，但是，如果该盘块区填满了，将不再是去做一个新的副本，而是直接进行第二次的分配，且该分配得到的盘块区（即所谓的扩展盘块区）不一定非得与初始分配得到的盘块区相连续。第二次分配得到的扩展盘块区自身也是连续的，不过其通常要小于初始分配得到的盘块区（即所谓的主盘块区）。还可以进行第三次、第四次的分配，但此类再次分配的次数一般应在某个不多的限度范围之内——如可设定为最多不超过 16 次左右。需要指出的是，这时的对随机存取的文件地址的计算就要有点复杂了。对于单一连续的文件而言，我们拿到对应记录或字节偏移量，就可以计算出存取位置距离文件首部盘块的偏移量。而现在我们还需要拥有一张表格，该表格填有各扩展盘块区的起始逻辑地址、起始物理地址以及相应的大小。我们计算出偏移量，然后查找该表格，从中找到包含该偏移量的那个扩展盘块区，进而计算出相对于该扩展盘块区开始位置的偏移量。相比于硬盘的速度，这还是微不足道的。扩展盘块区并不是一种特别新的解决方案，例如，至少早在 20 世纪 60 年代末期，由 IBM 公司开发的 OS/360 系统中已经使用过这种方法。

连续分配方案存在空间浪费的几种情况。第一种情况是由我们可以访问的最小空间部

274

分是扇区的事实所造成的，而且我们经常是以盘块而非扇区来进行分配和监测空间的，从而进一步加剧了相关问题。因此，我们可能会按由4个扇区组成的盘块来实施分配，但大多数情况下，我们并不需要此类分配得到的全部空间。有时，我们会正好填满最后一个盘块，但有时我们仅仅需要最后一个盘块的一个字节而已，于是平均下来我们只使用了其中的一半空间。这种由分配粒度所造成的尚未利用的空间被称为**内部碎片**（internal fragmentation）。我们在第10章中讨论主存分配的时候，曾遇到过完全相同的问题。除非我们有很多非常小的文件，否则，以现在的磁盘驱动器的空间大小和成本来讲，磁盘驱动器上的内部碎片并不总会被摆到非常显要的重视层级上来。

更为紧要的是**外部碎片化**（external fragmentation）的问题。同样，这一问题在关于内存管理的第10章曾进行过讨论。当我们接近于填满磁盘的时候，外部碎片这个问题就会凸现出来。伴随我们对连续文件的分配和释放，我们往往会趋向于把整个空闲空间切割开来，因为我们总是从较大的空闲空间中取出一片连续的空闲空间来。直到最后，残留的空闲连续空间都变得太小而全都无法独立满足我们想要进行的下一次分配需求，即使这时空闲空间的总量能够满足分配需求，我们也只能望洋兴叹。例如，在图12-13中，根据图中显示的空间大小，我们大概还有约2000个盘块大小的空间，但是由于该空闲空间被划分为两个部分，所以我们就不能够分配给一个那么大的文件，尽管我们拥有足够多的空闲空间可以分配给这样的文件。此类问题的解决方法有些麻烦，称之为**碎片整理**（defragmentation）。其基本思想是把一些文件移到能够容下它们的残留空间里，从而腾挪出较大的连续空闲空间来满足我们想要分配的文件。该项技术在第10章中已进行了非常详尽全面的描述，故而在不再赘述。第三种类型的“碎片”是在介绍程序员会为文件分配多于其真正所需大小的空间的时候曾经讨论过的**程序员碎片**。不过，就程序员碎片而言，与其说它是一种技术问题，倒不如说它是一种社会问题。但是，该问题是因为很难使一个连续的文件变大而演变出来的，所以需要被提到。

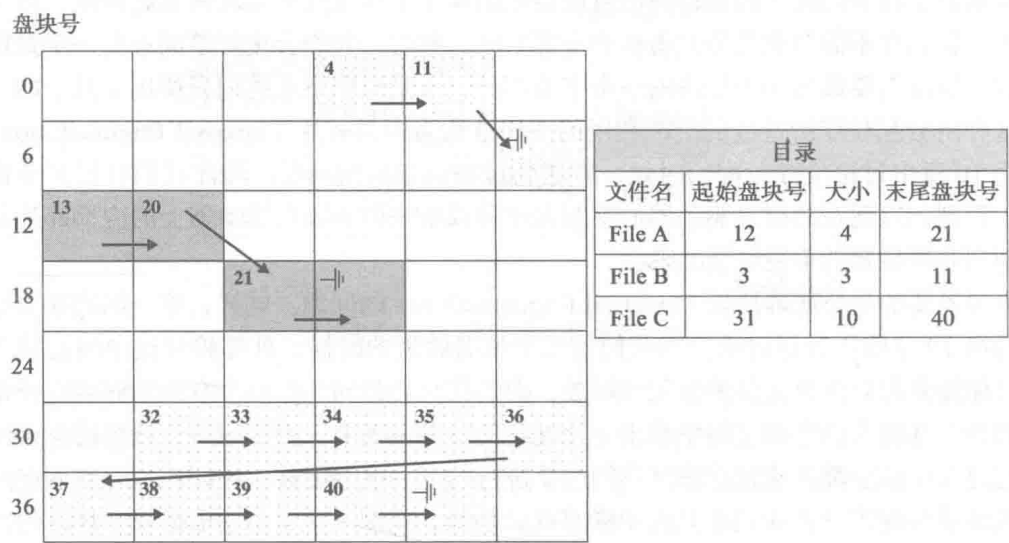
12.5.2 链接分配

第二种常见的文件分配机制是链表。这种机制就像主存中的链表结构一样，不过这里链接的元素总是大小相同的，即均为一个盘块的大小。每一个盘块将包含其所在同一文件的下一盘块的起始扇区地址。于是，这种链接机制的缺点之一就是每一盘块都会消耗一部分空间来存放链接指针。在最坏的情况下，我们拥有一个可能为512字节大小的单独的扇区，而一个指针可能占用了4字节，由此推算，空间浪费额度还不到1%。如果盘块大于一个扇区，那么花费在链接指针上的开销会更少。图12-14给出了此类结构的一个示例。

275

这种链表分配机制还有另外一个缺点，即针对此类文件实现随机存取方法有些困难。尽管如此，并非没有可能。认真分析一下图12-14中给定的文件，针对相关文件提供随机访问所需做的全部事情就是把连续分配一节所讨论的扩展盘块区的思想进一步拓展开来。我们只是需要在内存中建立一张表，以给定对应文件分配得到的每个盘块的起始位置的指针。虽然，对于一个很大的文件，这张表可能比较大，而且循着整个链去构建这张表可能会花费一些时间，但是在大多数情况下，这却可能是一种切实可行的解决方案。如果文件很长时间都不会打开，那么建立和存储这张表所需的空间和时间开销可能有些过高。但如果要对文件进行大量的随机访问，而且该文件也不是太大，那么这种方案还是蛮实用的。还有，当文件第一次被打开时，我们未必一定需要循着整个链走上一遍。只有当对记录的引用使我们需要访

问文件中我们尚未读取对应指针的部分的时候，我们才可能循着链来填写这张表格。



我们可以看到，在图 12-14 中，描述链接文件的目录项包含有指向文件起始盘块的一个指针以及用盘块数表示的文件长度，同时还有一个指向链表中最后一个盘块的指针。乍一看，存储指向文件末尾的指针似乎没有什么必要，而且事实上也没有必要，因为我们总可以沿着链表中的指针找到末尾。但是，这样设置却具有两个非常实际的理由。第一个理由是，有时我们想以“添加”模式来打开文件——也就是说，我们正好想把数据添加到文件的末尾。日志文件就是践行这种操作的一个很好的例子。这样设置，总能够更快地直接到达链表的末尾。第二个理由与冗余有关。对于文件系统的元数据而言，有一些冗余总是好的。那样，当发生问题时，我们运行来修复文件系统的实用例程就会有更加精确的指标，从而可以规划出正确的行动方案。

[276]

关于链接文件，其好的方面在于，我们不用去担心程序员碎片问题。因为扩展一个链接文件是非常简单的事情，没有任何压力一开始就得去为文件分配太多的初始空间。

在关于连续文件分配的一节，我们曾经讨论过，当有足够的空闲空间可以满足分配需求但可用空间并不连续的情况下，就需要对空间展开紧凑处理。我们还提到相关过程有时被称作碎片整合。或许有些令人惊讶，链接文件也面临着所说的结构性问题，而碎片整合这个术语应用到这个问题可能会更合适些。链接文件的结构可以看成是采用连续的扩展盘块区的极端情况，也就是说，其扩展盘块区也就一个盘块的大小。链接文件所存在的问题是，随着文件长度的增大，“下一可用的”盘块可能位于磁盘上的任何地方。为此，链表就可能跟着文件变大时可用盘块的位置而在磁盘上来来回回跳来跳去。此类极端的分配示例如图 12-15 所示。如果让一道完成很多输入/输出操作但却没有多少计算处理操作的程序来处理这样的文件，相关开销可能会很大。重新整理所有的文件，从而使分配给每个文件的盘块是有序和连续的，这一过程称为碎片整合。碎片整合可以有效地提高文件处理的速度。就像早先所提到的，某些系统往往同时支持连续文件分配和链接分配。许多现代的操作系统一般都支持这两种类型的文件分配，所以其结果是它们既有外部碎片的问题，也有随机链表的问题。因此，在此类操作系统中，通过碎片整合来应对相关这两个问题都可以发挥一定的作用。

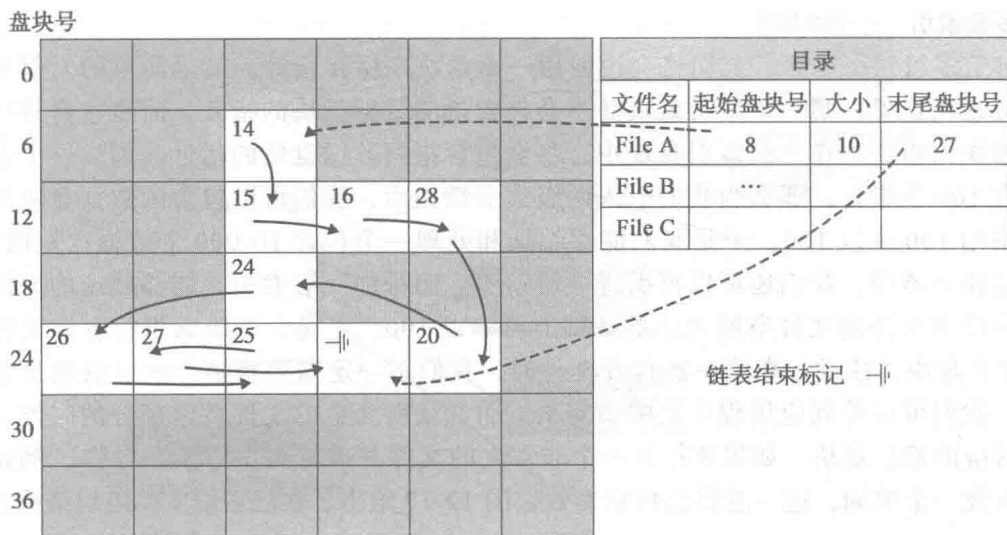


图 12-15 碎片化文件示例

277

12.5.3 索引分配

就像空闲空间监测可以采用索引方法一样，文件结构的记录和描述也可以采用一种类似的机制来实现。从最简单的观点来看，索引文件结构有点像一个链表，只是我们分配了一个单独的索引盘块来专门存放指针，而不是将指针分散放置在每一个数据盘块中，相应的分配方式称为索引分配。图 12-16 就举例说明了某文件（例如 File A）中的许多盘块是通过一个索引盘块来存放和标示，而不是各自单独分别进行链接的。就像索引式空闲空间监测机制一样，在最简单的实现方案中，我们仅限于唯一的一个索引盘块。该限定将会限制文件的大小，因为盘块的大小是固定的，于是索引盘块能够持有的指针数只能在受限的最多盘块号数量之内。要想扩展相关机制和消除这个限制，一般可采用两种方法。具体而言，我们可以使用多级索引，类似于我们对内存页表的处理方式，或者我们也可以把索引盘块自身链接起来形成一个链表。

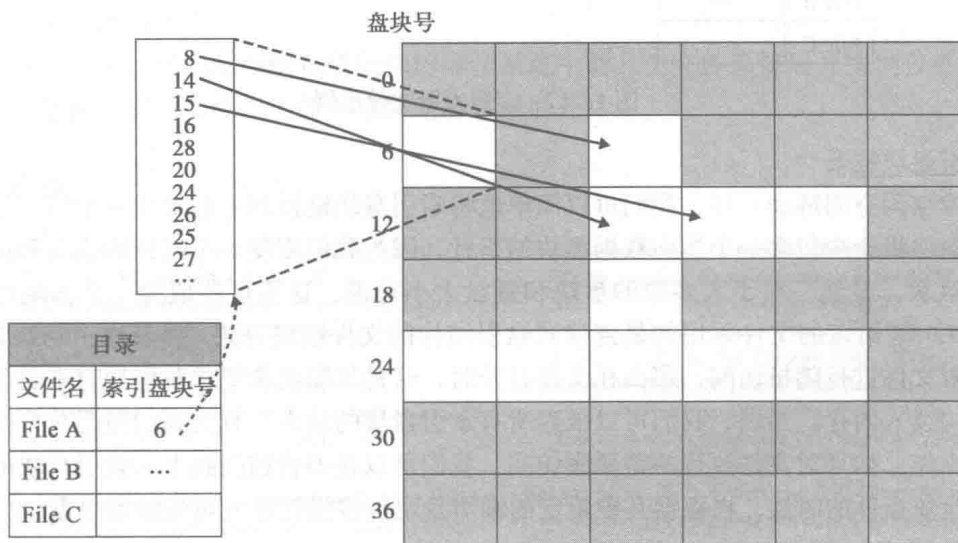


图 12-16 索引文件示例

多级索引

[278]

对于多级索引而言，我们将会再使用一个盘块来保存指针，就像简单的索引结构一样。但是在这里，第一个索引盘块并不会包含指向数据盘块的指针，而是包含指向二级索引盘块的指针。在二级索引盘块中，将会包含指向数据盘块的指针。如果一个索引盘块包含 100 个指针，那么当我们引入两级索引结构后，我们可以包含的数据盘块的指针数应是用 100 乘以 100。于是我们能够编址和处理一个包含 10 000 个数据盘块的文件。如果这还不够用，我们还可以再引进一层索引。而每加一层索引，能够处理的文件大小将会在原有支持的文件空间大小的基础上再乘以 100。于是，三级索引将容许文件包含 100 万个盘块。注意，在第一次打开文件时，我们不一定需要将整个索引盘块集都读入内存。我们可以等到应用程序试图访问某索引盘块所涵盖的文件数据部分的时候，再来读取对应的索引盘块。如果要打开一个非常大的文件并执行简短的读取操作，例如在字典中查找一个单词，这一着将会特别奏效。图 12-17 给出了多级索引文件组织结构的一个例子。

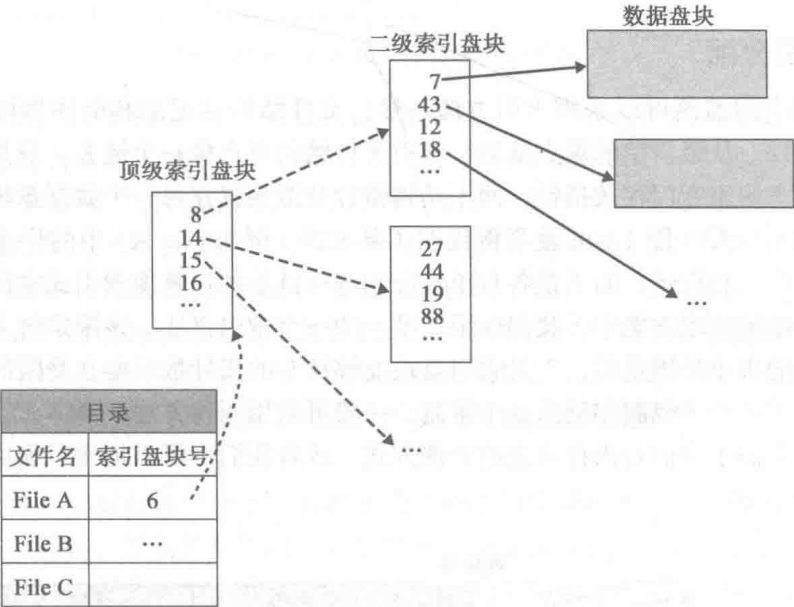


图 12-17 多级索引文件示例

索引盘块链表

[279]

就像空闲空间链表一样，我们可以简单地将索引盘块链接到一起形成一个链。于是，每个索引盘块将会少包含一个指向数据盘块的指针，因为我们需要一个指针来指向和访问下一个索引盘块。显然，对于大多数的盘块和磁盘大小来说，这不可能成为一个影响严重的因素。图 12-18 给定的文件采用的是链接式索引结构的文件组织方式（译者注：例如 File A）。如果要对文件进行随机访问，那么在文件打开时，这种机制将会要求我们循着索引盘块链把索引盘块读入内存。当然，我们可以推迟所有索引盘块的读入，直到我们需要时再执行相应的读取操作。如果对文件执行的是顺序访问，我们可以在即将访问到上一索引盘块的最后一个数据盘块指针的时候，再根据其中给定的索引盘块链接指针读入对应的索引盘块。

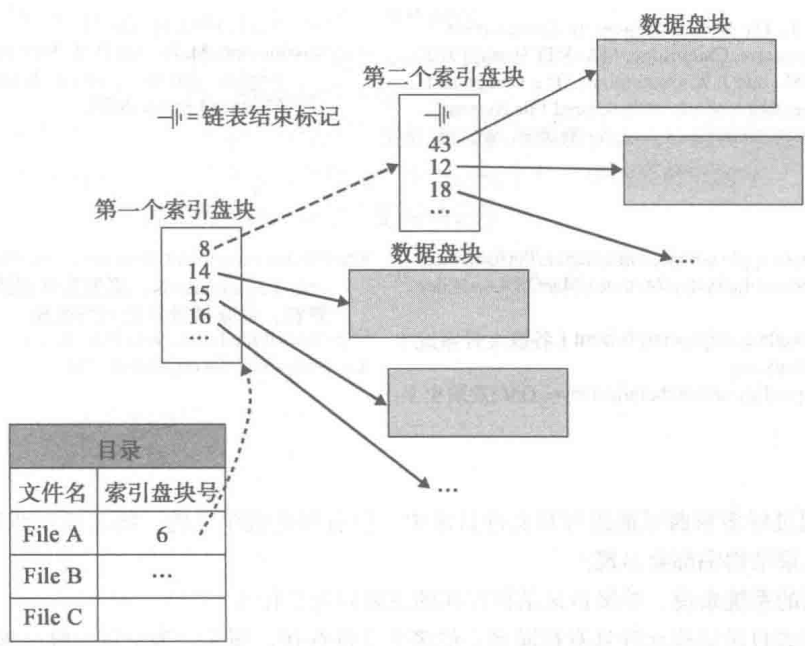


图 12-18 基于索引盘块链表的文件结构示例

12.6 小结

文件是操作系统提供的一种重要的抽象。早在计算机出现之前，文件就已经在使用了，所以文件为大家所熟知。程序设计人员不想去操心硬件的东西，他们想要考虑的是数据的集合。在计算机系统中，数据的集合就是文件。在本章中，我们讨论了文件系统的本质，然后介绍了操作系统的文件系统的基本思想。现代计算机系统拥有很多文件，故而可能需要组织文件，以便我们可以找到相关的东西。我们讨论了文件系统目录。不同的应用程序需要不同的存取数据的方法，因此我们描述了可以提供给应用程序用来存取文件中的数据各式各样的方法。文件系统需要记录和监测整个空间中有哪些空间当前是空闲的。我们探索了用于监测这些空闲空间的各种不同的结构。我们还就文件自身结构的话题进行了阐述，并讨论了不同方法的权衡问题。

在下一章中，我们将会就众所周知的操作系统中的几个文件系统展开案例分析，同时还会就关于操作系统的文件系统的一些其他的话题展开讨论。

参考文献

Beck, M., et al., *Linux Kernel Programming*, 3rd ed., Reading, MA: Addison-Wesley, 2002.

Bovet, D. P., and M. Cesate, *Understanding the Linux Kernel*, 2nd ed., Sebastopol, CA: O'Reilly & Associates, Inc., 2003.

Golden, D., and M. Pechura, "The Structure of Microcomputer File Systems," *Communications of the ACM*, Vol. 29, No. 3, March 1986, pp. 222–230.

Koch, P. D. L., "Disk File Allocation Based on the Buddy System," *ACM Transactions on Computer Systems*, Vol. 5, No. 4, November 1987, pp. 352–370.

Larson, P., and A. Kajla, "File Organization: Implementation of a Method Guaranteeing Retrieval in One Access," *Communications of the ACM*, Vol. 27, No. 7, July 1984, pp. 670–677.

Livadas, P. E., *File Structures, Theory and Practice*. Englewood Cliffs, NJ: Prentice Hall, 1990.

McKusick, M. K., W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181–197.

Nelson, M. N., B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 134–154.

Organick, E. I., *The Multics System: An Examination of Its Structure*. Cambridge, MA: MIT Press, 1972.
 Rosenblum, M., and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, Vol. 10,

No. 1, 1992, pp. 26–52.
 Russinovich, M. E., and D. A. Solomon, *Microsoft Windows Internals*, 4th ed., Redmond, WA: Microsoft Press, 2005.

网上资源

<http://developer.apple.com/documentation/Performance/Conceptual/FileSystem/Articles/MacOSXAndFiles.html>
<http://labs.google.com/papers/gfs.html> (谷歌文件系统)
<http://www.linux.org>
<http://pages.prodigy.net/michaln/history/> (OS/2发展史)

<http://technet.microsoft.com/en-us/sysinternals/default.aspx> (Sysinternals, 原先是外围的技术参考资料, 后来被微软公司所收购)
http://en.wikipedia.org/wiki/File_system
<http://en.wikipedia.org/wiki/CP/M>

习题

- 12.1 我们提到过好多东西可能出现在文件目录中, 但有些是很罕见的。哪几样东西最可能在每种操作系统目录结构中都会出现?
- 12.2 对于今天的系统来说, 单级目录结构存在的主要问题是什么?
- 12.3 既然层次式目录结构允许具有相同名字多个文件存在, 那么, 为了可以唯一地指定这些文件, 我们应该如何引用它们呢?
- 12.4 什么类型的问题促成了对别名的使用呢?
- 12.5 为了改善搜索时间效率, 应该如何优化目录内部组织呢?
- 12.6 为什么操作系统通常会提供特殊的调用来访问目录项呢?
- 12.7 关于诸如 `erase <文件名>` 这样的命令, 工作目录会起到什么样的效果呢?
- 12.8 如果通过顺序存取方法来处理一个文件, 对于一次读操作, 当前记录指针会发生什么样的变化呢?
- 12.9 采用随机存取的一个应用程序不能只是请求操作系统访问下一条记录, 而总是必须指定相应的记录号。这种说法是否正确?
- 12.10 对于索引顺序存取来说, 每条记录的主关键字字段必须包含唯一的键值。这种说法是否正确?
- 12.11 原始存取提供了什么样的服务呢?
- 12.12 在文件系统中, 两种基本的空闲空间记录监测机制是什么?
- 12.13 对于我们介绍的空闲空间记录监测机制, 其中的一种机制拥有两种宽泛的改进方案, 且其中一种改进方案引入了一项其他的改进措施。这些改进对策均致力于与该空闲空间记录监测机制相关联的重要性能问题的缓解。这里说的是哪种空闲空间记录监测机制? 我们所关注的问题是什么呢?
- 12.14 在前一个问题中所提到的“其他的改进措施”是一种也可以应用在其他空闲空间机制及其改进方面的技术, 这种技术具体指的是什么呢?
- 12.15 在文件系统中, 用于文件空间分配的三种基本机制是什么?
- 12.16 有一种文件空间分配机制对于随机存取文件来说是最方便的, 它是哪一种呢?
- 12.17 (和上题答案) 相同的分配机制很难增大文件的长度。这个问题带来了一个间接的问题, 那么这个问题是什么呢?
- 12.18 (和上题答案) 相同的分配机制很难增大文件的长度。为此, 我们描述了允许增大文件长度的该基本机制的一个变种。该变种是什么样的呢?
- 12.19 什么是内部碎片? 为什么在大多数情况下内部碎片不再是一个问题了?
- 12.20 什么是外部碎片? 对于某些操作系统而言, 为什么外部碎片的问题比内部碎片更厉害?
- 12.21 我们曾提到“程序员碎片”的问题, 并且声称其中一种文件分配机制没有这类问题。是哪一种

文件分配机制呢？为什么它不会有程序员碎片问题？

12.22 (和上题答案) 相同的机制存在一个严重的缺陷，具体是什么呢？

12.23 简要描述链接文件的碎片整合要做哪些事情。

12.24 由于能够存储在唯一的一个索引盘块中的指针的数量是有限的，所以最简单的索引文件分配方法限制了文件的大小。为了突破这种限制，本书讨论了什么样的两种机制？

12.25 对于随机文件访问来说，哪些机制可能会更好地运作？

282

文件系统实例及更多功能

13.1 引言

在第 12 章中，我们阐述了文件系统的概念以及文件系统是如何融入操作系统中的。我们介绍了关于文件存储和空闲空间监测的许多可能的备选机制，而实际的文件系统的设计人员必须对系统将要包含的机制做出选择。我们将会看到，现代操作系统使用了我们所描述的所有技术，但是没有任何一种操作系统是完全按照我们所描述的方式来使用这些技术的。

在第 13.2 节中，我们将会围绕现代操作系统中文件系统的相关设计给出一些案例研究。在此基础上，我们将会讨论文件系统和文件处理相关的若干其他主题。进一步说，我们将在第 13.3 节讨论挂载一个文件系统并将其中的有关信息提供给应用程序的基本原理，而在第 13.4 节我们将会剖析虚拟文件系统和相关概念背后的根源，同时，在第 13.5 节我们将会阐明内存映射文件的目的。通常情况下，操作系统会提供大量的实用程序以实现文件系统信息的标准化操作。为此，我们在第 13.6 节讨论其中的一些实用程序。另外，我们在第 13.7 节对构建更可靠文件系统的事务型文件系统技术（即基于日志的文件系统技术）的重要概念进行阐述。最后，我们在第 13.8 节对全章内容进行归纳总结。

283

13.2 实例研究

一般来说，实际的操作系统往往会综合运用第 12 章所描述的基本技术。例如，我们曾经提到过，某些操作系统既支持连续文件分配（contiguous file allocation），又支持链接文件分配（chained file allocation，或称为链式文件分配）。在下面的几个小节中，我们将对文件系统展开实例研究，即近距离地审视一些现代的文件系统，并探讨它们的实现机理。

13.2.1 FAT 文件系统

我们仔细研究的第一种文件系统是一种链接系统的改版。需要强调的是，这种文件系统没有在每个数据盘块中包含指向下一个数据盘块的指针，而是把那些指针都统一存放在一个单独的表中。这种文件系统曾经用在原来的微软 DOS 操作系统中，并按用来存放相关指针的表的区域的名称，即文件分配表（File Allocation Table, FAT）进行了命名。在 FAT 文件系统中，文件分配表并没有保存在用于存储数据的区域中。进一步说，文件分配表位于磁盘上刚好在引导块之后的一个单独的区域中。该表中将会顺序给出数据区中的每个盘块所对应的一个盘块指针。如果有关盘块尚未分配给一个文件，那么对应表项中的指针将设置为零。而如果某个盘块是一个文件的一部分，那么对应表项通常情况下应包含指向同一文件中的下一个盘块的指针。特别地，如果有关盘块是所在文件中的最后一个盘块，那么对应表项将包含一个特殊的指针值，用来指示对应盘块已是文件盘块列表的结尾盘块。图 13-1 显示了一

个 FAT 文件系统及其中的两个文件。图中，我们把文件结束标志标记为 FFFFFFFF。

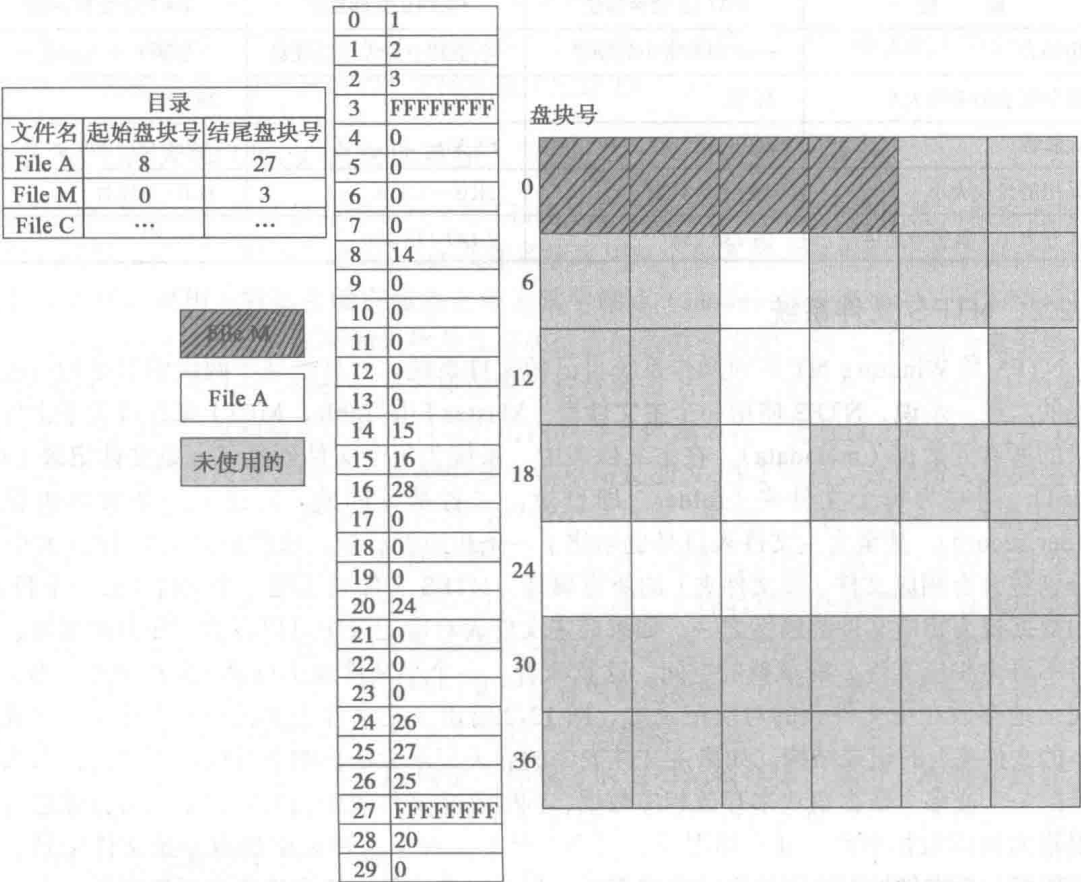


图 13-1 FAT 文件系统及其文件示例

关于文件分配表机制，有一些很有意思的事情需要注意。首先，其没有设立单独的机制来跟踪空闲空间。空闲空间的相关盘块在文件分配表中的相应表项均为 0（译者注：即所含盘块指针取值为 0）。其次，利用文件分配表很容易为一个文件分配连续的空间。就像采用位图空闲空间机制一样，为了找到一组连续的空闲盘块而必须完成的全部事情就是，扫描文件分配表并查找一个连续的零指针串。同时，采用文件分配表方式也很容易实现单一扇区的分配，能够支持面向链接文件访问的单个盘块的分配。

最初的 FAT 文件系统设计是针对软盘驱动器而创建的，故而有关的指针十分短小，对应系统后来被称为 FAT12 文件系统。这种文件系统曾经被用在早期的小硬盘上。然而，磁盘容量迅速扩增，到后来，即使不是用单个扇区而是采用较大的盘块（译者注：盘块在 FAT 系列文件系统中又称为簇，即 cluster，由若干扇区组成）进行分配，也无法覆盖到整个磁盘空间。为此，一种新的文件系统被设计出来，其与 FAT12 文件系统非常相似，但是采用了较大的指针，该系统被称为 FAT16 文件系统。鉴于当时的计算机拥有 16 位的字长大小，所以 FAT16 文件系统所设定的指针大小在那个时代看来是开展文件系统设计的一个相当合理的尺寸。尽管如此，这一大小还是相当受限的，因此 FAT16 文件系统设计方案后来被 FAT32 文件系统所取代。关于这三种文件系统的概括总结如表 13-1 所示。

表 13-1 各种 FAT 文件系统的比较

属 性	FAT12 文件系统	FAT16 文件系统	FAT32 文件系统
适用场合	软盘和非常小的硬盘	小型到中等大小的硬盘	中型到非常大的硬盘
文件分配表的表项大小	12 位	16 位	28 位
最大簇数	4096	65 536	>260 000 000
所采用的盘块大小	0.5KB~4KB	2KB~32KB	4KB~32KB
最大卷大小（以字节为单位）	16 736 256	2 147 133 200	大约 2 ⁴¹

13.2.2 NTFS 文件系统

NTFS 是 Windows NT 系列操作系统自己的文件系统，并且是基于两级索引结构改进而形成的。进一步说，NTFS 使用一个主文件表（Master File Table, MFT）来存放关于文件和目录的所有元数据（metadata）。在主文件表中，系统为每个文件创建了一条文件记录（file record），同时为每个文件夹（folder，即目录，二者是等同的）创建了一条文件夹记录（folder record），甚至为主文件表自身也创建了一条相应的记录。这些记录均为 1KB 大小[⊖]，且分别包含有相应文件（或文件夹）的所有属性。NTFS 文件系统把一个文件（或一个目录）中的数据视为相应文件的属性之一。如果在主文件表对应记录中可以容纳下所有的属性，那么将不再为相应文件分配单独的空间。这意味着，一个小文件或小目录（大约 900 字节）将会完全地存放在主文件表的对应记录中。图 13-2 给出了主文件表中关于一个小文件（或一个小的文件夹）的记录结构。如果主文件表中的有关记录无法容纳下对应数据属性，那么将会分配一个或多个数据盘块来存放相应数据，同时主文件表中的对应记录上将会建立一个索引指向相应数据盘块。通常情况下，每个文件只会在主文件表中拥有一条文件记录。然而，如果一个文件拥有许多属性或非常零散，那么对应文件在主文件表中可能需要一条以上的记录。在这种情况下，相应文件在主文件表中的第一条记录被称为文件基本记录，存放有相应文件所需的文件扩展记录的位置。

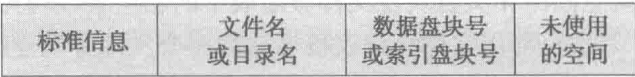


图 13-2 NTFS 文件系统的主文件表中关于小文件或小目录的记录结构

文件夹（或目录）记录包含有索引信息。小文件夹的记录完全驻留在主文件表结构中。而大的目录则组织成 B 树结构，并且该结构带有一系列指针指向若干外部的簇，簇中包含有不能包含在主文件表结构中的目录项。当 NTFS 文件系统在一个非常大的文件夹中保存有若干文件的时候，B 树结构的好处是显而易见的。进一步说，B 树结构把相似的文件名集中到一起、归为一组和放到一个盘块中，这样系统仅仅需要搜索包含对应文件的分组即可，从而将会使文件查找所需的磁盘访问工作量最小化。关于 NTFS 文件系统的其他要点包括：

- 使用一个位图来记录空闲空间。
- 在后来的版本中支持可变的盘块（簇）大小。
- 支持整个文件系统、目录、子树或单个文件的压缩。
- 支持整个文件系统、目录、子树或单个文件的加密。

⊖ 事实上，NTFS 系统的详细资料属于专利性质。这里使用的数字得到大家的普遍认可，但不一定非常准确。

284
285

- 支持软件 RAID 1 和软件 RAID 5（见第 14 章第 14.6 节）。
- 保持关于系统不再使用的坏簇的一个单独的映射。
- 针对仅仅包含二进制零（即 nulls）的一个文件，不会写到磁盘（较大的）部分。
- 一种事务型（基于日志的）文件系统（见第 13.7 节）。

13.2.3 UNIX 和 Linux 的文件系统

UNIX 及其衍生的许多操作系统（如 Linux）均支持多种不同的文件系统，但它们对 ext 文件系统的支持则是相当普遍的。ext 文件系统采用某个版本的多级索引方案来保存文件相关的元数据，而相关数据存储在磁盘上称为索引结点（inode）的一张表中。在 UNIX 目录中，每个表项只包含有对应项的名称及其存放位置的数值型引用。该引用被称为索引结点编号（i-number 或 inode number），是对所谓索引结点列表（i-list，或称为索引结点集）的索引。索引结点列表的位置和格式以及索引结点的内容等具体细节在一定程度上取决于特定 UNIX 版本或其衍生操作系统，同时，典型的索引结点信息如表 13-2 所示。这里所关注的是在索引结点里有什么信息以及哪些信息没有放在索引结点里。首先，文件名没有放进索引结点。UNIX 系统允许文件拥有别名，这意味着一个以上的目录项可以指向同一个文件。另外，对同一文件的不同引用并不要求使用相同的名称。因此，文件名存放在目录中，且相应的目录项指向包含有对应文件所有其他元数据的索引结点。其次，索引结点中有一项给出了指向对应文件的目录项的数量。当一个文件的某个目录项被删除时，引用计数将会递减（即做减一操作），但是文件本身直到引用计数变为零时才会被删除。

UNIX 文件系统的索引结点结构是一种混合的索引结构变体。其中，有许多指针直接指向数据盘块，这类指针的数量不尽相同，但通常在 10~13 之间。当文件被打开时，对应索引结点会加载进主存，故而，如果相应文件非常小，那么指向前几个盘块的指针已经获得和可以直接使用。但是，如果文件足够大并且所需盘块数比直接指针所指向的盘块的数量还要大，那么下一个指针就应当是指向一个单级索引盘块的指针。如果系统采用的是 4KB 大小的盘块，那么这个一级索引盘块里最多可以包含对应文件直接指针所指盘块之外的 1024 个盘块。而如果所有这些空间也被用完，那么还需用索引结点里的下一项，即指向一个两级索引盘块的指针。该索引盘块所包含的指针并没有指向数据盘块，而是指向其他的索引盘块。因此该索引盘块将会寻址 1024 个索引盘块，从而总共寻址会 100 万个数据盘块。而如果这也不足以容纳下有关文件的数据，则索引结点中的下一项即指向一个三级索引盘块的指针就必须得用上了。还是按照刚才所描述的 4KB 大小的盘块来进行推算，那么该三级索引结构就可以寻址超过 4TB 的文件空间。

当一个 UNIX 文件系统被初始化时，将会建立

表 13-2 典型的 UNIX 索引结点内容

文件类型
访问权限——读、写，等等
文件被引用的目录项的计数
所有者
所有者组别
创建时的日期和时间
最后一次访问的日期和时间
最后一次修改的日期和时间
大小
数据盘块指针 1
数据盘块指针 2
.....
数据盘块指针 10（有时为数据盘块指针 13）
单级索引盘块指针
两级索引盘块指针
三级索引盘块指针

286

287

一个与对应磁盘分区大小及所采用盘块大小相适应的恰当大小的索引结点列表。于是，一系列空的索引结点将被创建并均匀分布在整个分区上。当为一个文件分配盘块的时候，首先会从靠近对应索引结点的可用盘块中进行选择。这种处理方式将有助于使分配给一个文件的所有盘块尽量是彼此相邻的。只要其他进程没有访问太多的位于驱动器其他地方的其他文件，那么这还将有利于使访问对应文件所需的查寻时间最小化。

13.3 挂载

有些时候，我们必须得和具有多重涵义的计算机科学方面的术语（称之为重载）打交道。挂载（mounting）就是这样的一个术语。实际上，这个术语的两种含义是相关的，但乍看起来，它们却似乎指的是不同类型的操作。其第一种含义指的是当包含有文件系统的一个磁盘驱动器分区即将被操作系统访问的时候所必须完成的相关事项。而其第二种含义则是指用来赋予用户指定位于一个远程目录上的文件的手段的过程。

13.3.1 本地文件系统挂载

在操作系统允许用户访问特定的文件系统之前，系统必须要完成某些事情。例如，读取用来描述分区的元数据，把空闲空间机制的某些部分（譬如预先分配的某些盘块等）读到内存，读取代表目录树根部的目录（即所谓的根目录），等等。这一过程被称为挂载。当安装操作系统时，往往会说明相应系统要访问的一些分区，而通常情况下这些分区将会在操作系统每次启动时被挂载，并且这些分区往往是本地的硬盘驱动器上的分区。然而，对于可移动介质而言，操作系统之间还是存在区别的。进一步说，操作系统一般按如下三种方式之一来处理可移动介质：1）当有关介质插入驱动器时，采用隐式挂载；2）当有关介质被首次访问时，采用隐式挂载；3）必须通过明确的挂载命令来启动挂载。

UNIX 及其大多数变种传统上均会使用最后的一种机制。也就是说，在用户给出特定的挂载命令之前，可移动介质是不能被访问的。由于微软 DOS 操作系统格式化的软盘是如此普遍甚至可以说是无处不在，所以这实际上便产生了一种有益的副作用。进一步说，这推动了支持用户明确指定一张软盘究竟要包含哪一种文件系统格式：UNIX、MS-DOS 还是 Mac。后来版本的 Linux 和 UNIX 已经开始尝试在有关介质插入时进行隐式挂载，且其就此所采用的术语为自动挂载（automounting）。微软 DOS 操作系统和 Windows 系列操作系统则一直采用当尝试第一次访问介质时的隐式挂载。在过去，Mac 操作系统通常是每当可移动介质插入驱动器时就触发相应的自动挂载。不过，鉴于 Mac 系列操作系统是基于 UNIX 操作系统的，所以其现在也采用了与 UNIX 相同的挂载方式。

在光盘（Compact Disc，CD）领域的情况则有所不同。在相当早期光盘开发时，大量的供应商聚集到一起召开会议，针对数据和音频光盘达成一致意见，并决定采用通用的格式。这种格式最终被指定为国际标准 ISO-9660。这种通用的格式意味着没有理由像 UNIX 系统那样来推迟挂载了，因此光盘往往是在插入时立即挂载的。这便允许有关操作系统检测光盘的格式（即音频格式、数据格式或混合格式），并且当此类光盘插入时还可按默认选项来予以启动执行。这意味着，如果用户做出了相应的选择，插入一张音频光盘将会按照用户的选择启动一个光盘音频播放器应用程序来播放这张光盘。同样，一张数据光盘可以包含一组在常见操作系统上针对光盘进行处理的相关指令。许多数据光盘经常会自动运行一个依赖于操作系统的脚本文件来启动光盘上的软件。

13.3.2 远程文件系统挂载

当请求操作系统来访问位于一台远程计算机上的文件系统时,也会发生类似的过程,不过相关细节却迥然不同。远程文件系统有可能是整个文件系统都被提供给用户使用,但更为可能的是一个文件系统的某些部分而不是整个文件系统的所有部分。其间必须要克服的一大困难在于,远程文件系统运行的平台可能与本地文件系统完全不同。具体来说,数据表示可能不同,文件命名规则可能不同,目录结构也可能不同,等等。为了克服和消除这些差异,我们必须围绕如何表示相关信息以及如何运用有关协议来交换信息等建立完善的规则。在大多数情况下,相关规则和协议往往是由一个平台的供应商所建立的允许对他们的系统进行互操作的事实上的规则。其他厂商将会创建软件包通过其他平台来访问这些系统。有时,这些规则还会成为开放的标准,就像网络文件系统(Network File System, NFS; 参见第 13.4.2 节)一样。然而有些时候,相关规则却是由其他厂商通过逆向工程而形成的。无论哪一种情况,都是远程系统在实施目录访问操作,但是相关信息必须要映射到客户端的操作系统环境中。例如,如果客户端是 Windows 系统,那么远程文件系统的象征就是相应的那个“驱动器盘符”。起初,这些盘符曾经在 DOS 系统中用来指示一个系统上的实际驱动器。而远程文件系统则采用了相同的约定,并且将会被指定一个当前没有用于本地资源的驱动器盘符。相比之下,UNIX 系统把所有的文件系统——包括像 `proc` 和 `dev` 之类的伪目录——都看作是一个树形结构。因此,对于 UNIX 之类的系统来说,对一个远程文件系统的映射仅仅需要在文件系统树形结构中添加(或更换)一个目录结点,且有关结点把自身标识成指向相应的远程资源。

从程序设计人员的角度来看,文件系统的远程挂载则是一种强大的工具。一般来说,有关程序并不知道本地文件(local file)和远程文件(remote file)之间有什么区别。换句话说,有关程序完全不需要进行任何修改,就能够跨越网络进行操作。遗憾的是,这并不总是一种明智之举。考虑一下数据库软件程序的案例,假定该程序正在访问一个通过网络远程挂载的数据库文件。当检索数据的索引时,数据库程序将会陷入跨网络的大量数据的读取和写入操作的“泥潭”之中。这如果是在流量负荷较低的快速的局域网(Local Area Network, LAN)当中,相关性能还可能是可以接受的,但如果是广域网(Wide Area Network, WAN)连接或者网络流量相当可观,那么文件系统的远程挂载可能并不是一种很好的方案。在这种情况下,更好的选择应当是在远程计算机上运行数据库程序,同时通过网络来发送 SQL (Structured Query Language, 结构化查询语言)命令,并且仅仅取回最终的结果。如果能够使用先前存放在远程服务器上的相关命令,甚至会更胜一筹。当然,不可能总是能够预测出所有的查询从而使它们事先存放在服务器上。有些时候,即席查询(ad hoc query)是必要的。

289

让文件系统中的一个结点成为对远程文件系统的引用可能会引发一个小小的问题。进一步说,路径名现在变得更加难以解析。如果在文件系统树中没有这些远程引用结点,解析一个路径名还是相当简单的。对于一个给定的路径如“/fred/work/expenses”,当操作系统解析该字符串时,每个“/”均表示进入本地文件系统中的一个目录。但是,如果有关结点可能代表树中的远程文件系统,那么相应文件系统必须逐级检查,以确认对应的结点是本地目录还是远程文件系统,进而执行相应的查找过程。

远程挂载的另一个问题是,两个不同的客户端可能在各自的本地文件系统中的不同位置挂载了同一个给定的远程目录。对于 Windows 系统来说,两个用户可能为相同的远程文件

系统指派了一个不同的驱动器盘符。而对于 UNIX 之类的操作系统来说，两个用户则可能把同一个远程文件系统挂载到了各自的本地文件系统的一个不同逻辑结点上。之后，如果某用户的机器上的一个进程向另一个用户的机器上的一个进程发送了一个路径名，那么第二台机器有可能无法找到对应的文件，因为它们的路径是不同的。有关管理人员可以通过定义标准化的挂载脚本，以在用户登录时加以运行并为公共访问资源的所有用户提供更加一致的路径命名机制，缓解乃至消除这一问题。

13.4 多文件系统和重定向

至于许多其他的情况，操作系统将会向应用程序接口呈现文件的抽象描述。有关程序不应该知道文件系统是什么样的。对于一个应用程序而言，如果其使用了错误的文件系统，很可能会发生性能方面的差异，但有关应用程序的编码不应该受到影响。这确实是一个系统工程的问题。如果有关应用程序被设计为随机访问一个文件，并且相应的文件系统支持随机访问，那么该应用程序应该不会意识到任何其他差异。在大多数系统中，往往要求操作系统支持多种不同的文件系统。最起码来说，如果不是由于其他的原因，其必要性就在于，对于不同的介质常常适合采用不同的文件系统。对于光盘来说，通常情况下只有 ISO-9660 标准格式需要考虑，尽管有一些非常早期的光盘曾经是以专有的格式创建的。对于软盘而言，几乎是默认的，也就是说，有关操作系统必须能够读取和写操作那些源自于微软 DOS 操作系统的 FAT12 格式的软盘。不过，Mac 格式和 UNIX 格式同样也被广泛使用。即使是硬盘驱动器，有时也会期望支持有关操作系统的原生格式之外的另一种格式。在一个系统升级到一种新的操作系统或者新版的相同操作系统时，这一需求通常就会发生。即便有关操作系统是相同的，但新版的操作系统也有可能随身携带了一种新的更棒的文件系统。尽管如此，当第一次执行升级过程时，有关文件系统仍然会维持那种旧的格式。之后，经常需要单独的步骤把旧的文件系统格式转换为新的格式。一个系统常常还需要包含两种不同的操作系统，并且根据当前的需要启动进入不同的操作系统。现在，甚至经常会看到一个虚拟机操作系统上同时运行着两个不同的客户机操作系统，并在不同的驱动器上支持不同的文件系统。另外还有人期望，不管当前使用的是什么操作系统，都应当能够访问有关磁盘驱动器上的所有各式各样的文件系统。基于所有这些原因的考量，故而操作系统需要支持许多不同的文件系统格式。

13.4.1 虚拟文件系统

正是为了透明地在同一个系统上同时支持多个文件系统，UNIX 系统开发人员创建了一种机制。这种机制被称为**虚拟文件系统**（Virtual File System，VFS）。虚拟文件系统是添加到 UNIX 系统且位于文件系统模块之上的单独的一层。实际上，虚拟文件系统是和支持不同文件系统的多个文件系统模块一起加载到系统中的。虚拟文件系统支持与现有文件系统相同的应用程序接口，故而有关应用程序无须进行修改。图 13-3 给出了虚拟文件系统引入前和引入后的应用程序与文件系统间接口的变化示意图，分别对应图 13-3a 和图 13-3b。当有请求传到虚拟文件系统层的时候，虚拟文件系统将会检查有关请求，通过查看文件系统树的相关结点，进而确定哪一个文件系统模块是对应于该请求的相应文件系统的正确的模块。然后，虚拟文件系统将会把该请求传递到相应的文件系统模块。当对应文件系统模块完成该请求任务时，其将会把控制权返还给虚拟文件系统模块，而后者再进一步把控制权返还给当初发出该请求的主调应用程序。

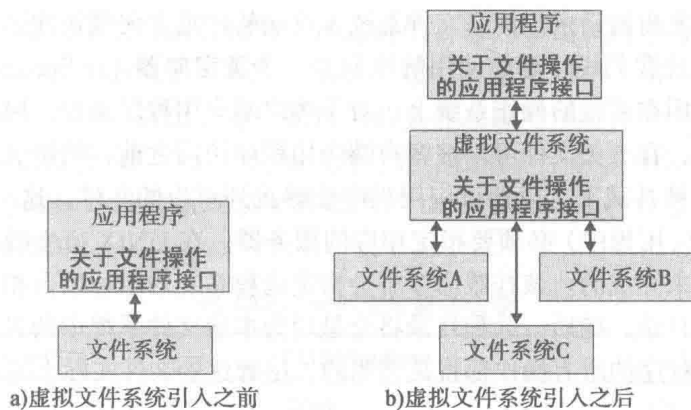


图 13-3 虚拟文件系统层的引入

13.4.2 网络文件系统

虚拟文件系统也被用于把有关的文件系统请求重定向到采用由 Sun 微系统公司开发的网络文件系统（Network File System，NFS）协议而远程挂载的相关驱动器上。这一过程曾经在之前的远程挂载的主题部分有所提及。图 13-4 显示了有关机制的工作原理。在图的上方给出的是相应的客户端系统。其间，应用程序通过标准的关于文件操作的应用程序接口而生成了相应的文件请求。接下来，虚拟文件系统意识到该请求要基于网络文件系统协议来访问另一个系统所提供的文件。在图的下方给出的网络文件系统服务器就是这里所说的另一个系统。于是，客户端系统的虚拟文件系统层便把该请求发给了网络文件系统客户端，而后者则利用远程过程调用机制来解决异构操作系统环境的相关问题，具体细节将会在第 17 章展开进一步的讨论。简单来说，就是客户端通过网络把该请求发送到网络文件系统服务器上运行的相应守护进程。然后，网络文件系统服务器守护进程拿到该请求后，将之发给服务器系统的虚拟文件系统层，进而根据有关请求（就像客户端是本地的一样）对相应文件进行了访问，相关结果数据最后被发回给客户端系统上运行的主调应用程序。

291

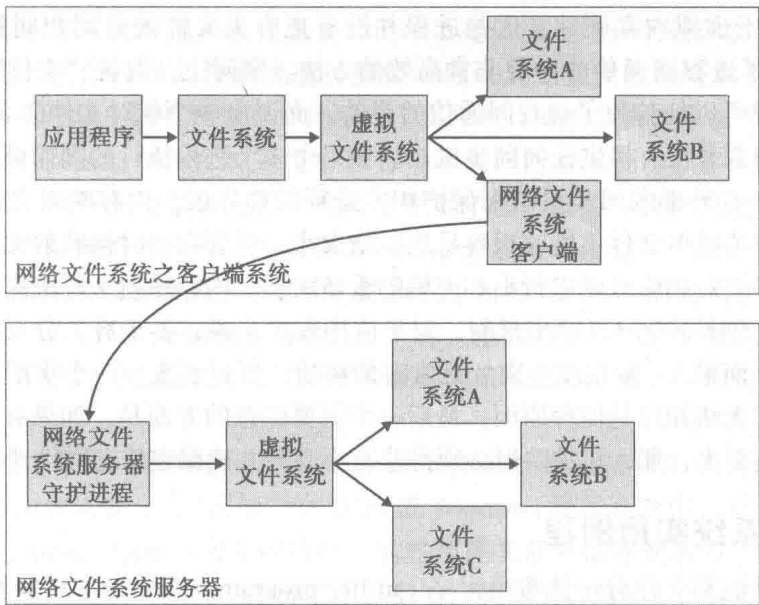


图 13-4 经由虚拟文件的网络文件系统服务过程

有关应用程序发出的请求由网络文件系统客户端软件重定向到正在运行网络文件系统服务器的系统上，为此我们说，当前使用的模型是一个**重定向器**（redirector）。这是一种常见的技术，也经常被用在其他的操作系统上。对于客户端应用程序来说，网络文件系统几乎是完全透明的。但是，在有关文件可以被客户端应用程序访问之前，网络文件系统服务器上的相关目录必须已经被挂载上，以便该应用程序能够找到对应的文件。这一步骤是不透明的，因为有关用户（或应用程序）必须要指定相应的服务器。在 UNIX 衍生的操作系统里，这是通过一条挂载命令来完成的。该挂载命令将会指定远程系统的名称以及相应系统上的一个目录，还有一个本地目录。随后，远程目录将会呈现为本地文件系统目录树的一部分，并且其对于任何应用程序的所有操作都将是透明的，尽管这些文件实际上是远程的。

13.5 内存映射文件

[292]

在许多操作系统里，还会发现另一种文件访问机制，即先前在第 11 章的高级内存技术部分曾经讨论过的内存映射文件。这一机制与文件的标准概念之间存在着非常大的区别。也正是因为它是如此的不同，所以它具有一些使其在某些特定情况下非常有用的特征。当一个应用程序使用一个内存映射文件时，其应告知操作系统要映射的文件的名称，然后操作系统就会创建一个按字节方式的从该文件的寻址空间到相应进程的逻辑寻址空间的映射。为此，关于内存映射文件的第一项非常有趣的特征是，相关文件系统的所有机制并没有被用来寻址对应空间。相反，有关应用程序把内存映射文件当作是一个大型数组来进行处理，并使用下标或指针运算来寻址整个文件。期间，虚拟内存管理器监测记录在物理内存中需要对应文件空间的哪些部分，跟踪发生更改的页面，并根据需要将这些页面写到磁盘上。这些机制利用了硬件支持，故而相比文件系统的有关机制要更为高效。或许你还记得，因为与分页机制之间的潜在的交互处理，正常的文件处理要么把进入内存的页面加以锁定故而抑制相应分页系统的性能，要么在执行输入/输出操作之前把输入/输出缓冲区复制到内核空间。然而，把有关文件映射到分页机制上就避免了这些问题，相关应用程序也无须再去执行任何内存分配。

内存映射文件的第二项很有趣的特征是允许多个进程同时通过内存映射到同一文件上。（有趣的是，由于虚拟内存硬件，这些进程并没有把有关文件映射到相同的逻辑内存地址上。）这便建立了进程间通信的一种非常高效的方法。实际上，有关“文件”并不一定真的存在。相关进程可以纯粹为了进程间通信的目的，而说出一个临时文件的名称。尽管如此，内存映射机制内部并没有提供任何同步机制。如果由多个进程执行的操作可能存在冲突，那么必须得使用一些外部的同步机制来保护相关进程的临界区。内存映射文件机制的另一个局限性在于，有关映射文件不能够很容易地扩增大小。尽管有些时候映射文件扩增也是可能的，但是要求必须对相应区域进行小心谨慎的重新映射。内存映射文件机制的第三个局限性在于，其没有提供异步输入/输出机制。对于应用程序来说，鉴于有关分页硬件是透明地进行读写操作的，而输入/输出发生阻塞是常有的事情，所以当发生一个页面故障时，有关进程将被阻塞但却无法知道是何种原因。最后一个需要注意的方面是，如果有关文件比可用的逻辑寻址空间还要大，那么相关映射必须得非常小心，并确保定位在文件地址空间。

13.6 文件系统实用例程

所有操作系统都会伴有一些实用程序（utility program，或称为实用例程），而且其中总会包括一组和文件系统打交道的程序。有些实用程序被设计成是在操作系统运行的同时来

使用的，它们包括诸如新建目录和删除文件之类的日常的事情。这些程序经常是通过命令行界面（command-line interface，又称为命令行接口）来运行的。表 13-3 就列出了 DOS/Windows 以及 UNIX 之类的系统中的一些常见的文件系统实用程序。大多数操作系统的后来的版本主要使用图形化用户界面，而相关命令往往没有一个名字是大多数用户都知道的。请注意，其中有些命令并没有出现在所有版本的 DOS/Windows 操作系统或者所有版本的 UNIX/Linux 操作系统中。

表 13-3 一些文件操作命令

实用例程的目的和作用	DOS/Windows	UNIX/Linux
改变文件操作权限	attrib	chmod
合并文件	backup	tar
列出目录中的文件	dir	ls
复制文件	copy	cp
删除文件	del	rm
删除文件系统子树	deltree	rm -R; rmdir
编辑文本文件	edit	vi
格式化磁盘	format	fdformat/mkfs
移动或重命名文件	move/rename	mv
列出文件内容	type	less
改变工作目录	cd	cd; chdir
按逐页方式查看文件的内容	more	more
创建或编辑磁盘分区	Fdisk	cfdisk, parted, etc.
创建一个新的目录	md, mkdir	mkdir

这些实用程序主要是由用户决定要做的事情。还有一些其他的实用例程也是必要的，但它们所做的事情尽管可能对用户也很重要，但却不是以满足用户的任何实际需要出发而执行的。进一步说，它们针对文件系统进行操作以确认其完整性或提高其性能。在 DOS 和 Windows 操作系统环境中就有两个这样的验证工具例程（即磁盘检查修复工具例程）分别称为 scandisk 和 checkdisk。类 UNIX 系统中也有一个类似的实用例程，称为 fsck。除了用来验证文件系统的一致性和完整性之外，这些实用程序还可以选择尝试修复它们所发现的故障。Windows 操作系统另外还有一个被称为 defrag 的磁盘碎片整理例程，能够对文件系统中的有关文件重新组织以提高系统的性能，其所针对的相关问题在前面关于链接文件分配的小节曾经讨论过。类 UNIX 系统的支持者往往声称，他们的文件系统从设计方案上就排除了碎片整理实用例程的需要。而此类实用程序未被推向市场的事实则表明，这种说法至少是相当精确的。

某些看起来很像实用例程的东西实际上是操作系统命令接口中的内置命令（built-in command，或称为内部命令）。例如，在 DOS 或 Windows 操作系统中，并没有可执行文件对应于 dir、del、time、type 等命令的执行。这些功能非常紧密地映射到了操作系统应用程序接口中的监管程序调用（Super Visor Call, SVC，又称为访管）上，故而命令解释器（对于 DOS 操作系统而言就是 command.com）拥有相应内置的功能函数，从而不再需要外部模

块。这便节省了磁盘空间以及把一个外部程序加载到内存的时间。

13.7 日志式文件系统

尽管系统故障极其罕见，但它们确实会发生。这也正是为什么要创建文件系统验证工具。当操作系统正常关机时，它会在文件系统里记录一条关于正常关机的说明。当操作系统启动时，它会检查确认上次系统是否正常关机或是否崩溃。传统上，如果有关系统崩溃了，那么在挂载文件系统之前，操作系统将会运行文件系统完整性检查程序。如果一个崩溃的系统是在被唯一的一个用户所使用，那么在故障发生时真正发生任何实际后果的可能性往往会比较低。即使是对于非常忙碌的服务器也常常不会产生什么高风险的损失。当然，与单用户系统上的崩溃相比，任何服务器故障所可能导致的问题将会影响到更多的用户。还有，对于单个用户而言，其在任何情况下都不会愿意失去任何东西。不管怎样，操作系统开发人员努力探索相关方法以确保文件系统能够更加健壮地应对故障。

当一个盘块已经被添加到一个文件并且该文件被关闭的时候，那么某些事项可能必须得付诸实施。譬如，我们一定得把包含有关数据盘块的记录写到磁盘上，我们还可能得查找下一个空闲盘块、更新空闲空间信息以标示相应盘块已被使用，以及更新对应文件的有关目录项以标示该文件最后一次写入的时间，等等。一般来说，我们希望所有这些信息的更新是以原子方式来实现的——或者所有信息都写到了硬盘上，或者其中的任何更新均未付诸实施。在应用程序中，我们称之为**事务处理**（transaction processing）。而操作系统中按照这样的一种方式运作的文件系统被称为**日志式文件系统**（log-based file system 或 log-structured file system 或 journaling file system，末者简记作 JFS）或**事务型文件系统**（transactional file system）。在此类系统中，任何时候有元数据被更新，系统都将首先写一条记录到日志文件以描述即将进行的所有更新操作。每当系统启动时，其都会检查日志文件，以确认是否存在悬而未决（即挂起而未完成）的事务。如果有这样的事务，系统就会检查确认有关事务的所有步骤是否均已成功地付诸实施。而如果没有全部付诸实施，那么系统就会尝试完成有关事务。如果可以完成，那么一切都很正常，意味着我们已经躲过了一劫。但如果有关事务由于某种原因无法完成，那么相关事务就会中止和取消。进一步说，我们将会主动丢弃要写入对应文件的最后一个盘块的数据，但是这样一来，文件系统就不会受到进一步的破坏。要知道，运行带有损坏的元数据的文件系统的后果将是灾难性的。

当然，没有什么事情是免费的，而我们为了日志式文件系统的安全性所需付出的代价就是性能上的损失。鉴于每次在我们写入元数据之前，都得花费时间来写入事务日志，所以我们将看到系统性能有所下降。此外，所记录的事务不一定包含有实际的用户数据，尽管某些操作系统确实在有关事务中包含有应用程序的相关数据。另一方面，如果一个系统拥有很多文件，那么一旦系统发生崩溃，在恢复系统操作之前，我们必须得首先执行一次完整的文件系统扫描来验证有关元数据的完整性，而在一台大型服务器上这可能差不多得耗费上好几个小时。因此，为了时时刻刻保持完整性，通常情况下，我们宁愿选择稍微降低系统的响应水平。这在单用户系统中尤其如此，其间，在用户敲击键盘或思考问题的同时，往往有大量空闲的处理器时间和磁盘时间可用来执行此类任务。鉴于此，最近几年开发的大多数文件系统都是事务型文件系统，其中包括用于 OS/2 操作系统的日志式文件系统、用于 Mac 操作系统的 HFS Plus、用于 Windows NT 系列操作系统的 NTFS 以及用于 Linux 操作系统的许多文件系统，包括 Ext3（第三代扩展文件系统）、ReiserFS、XFS 和 JFS。

13.8 小结

在第12章中,我们讨论了文件系统的概念以及它们是如何融入操作系统里的,包括许多可能的备选设计方案。对于真实的文件系统而言,则往往反映了其中所包含的相关机制的设计选择。为此,我们研究了现代操作系统中的文件系统的几个案例。这些简明扼要的概述说明了一些现代的操作系统是如何运用前一章所讨论的相关机制的。接下来,我们从挂载文件系统开始,依次讨论了与文件系统相关的其他问题。我们围绕诸如虚拟文件系统和相关概念背后的原因以及内存映射文件的目的等进行了专题讨论。我们介绍了操作系统必须提供和用来管理文件系统信息的一些实用程序。之后,我们还阐述了事务型文件系统或日志式文件系统技术背后的主要思想,正是相关技术才铸就了更为可靠的文件系统。

在下一章中,我们将介绍低级的输入/输出系统,其中主要是磁盘操作调度。

295

参考文献

- Larson, P., and A. Kajla, "File Organization: Implementation of a Method Guaranteeing Retrieval in One Access," *Communications of the ACM*, Vol. 27, No. 7, July 1984, pp. 670-677.
- McKusick, M. K., W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181-197.
- Nelson, M. N., B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 134-154.
- Organick, E. I., *The Multics System: An Examination of Its Structure*. Cambridge, MA: MIT Press, 1972.
- Sandberg, R., et al., "Design and Implementation of the Sun Network File System," *Proceedings of the USENIX 1985 Summer Conference*, June 1985, pp. 119-130.
- Sandberg, R., *The Sun Network File System: Design, Implementation and Experience*. Mountain View, CA: Sun Microsystems, Inc., 1987.

网上资源

- <http://www.linux.org>
- <http://technet.microsoft.com/en-us/sysinternals/default.aspx> (Sysinternals, 原先是外围的技术参考资料, 后来被微软公司所收购)
- <http://en.wikipedia.org/wiki/CP/M>
- http://en.wikipedia.org/wiki/Virtual_file_system
- http://www.yolinux.com/TUTORIALS/unix_for_dos_users.html (关于DOS/Windows和UNIX/Linux两类操作系统的命令的综合比较)

习题

- 13.1 为什么 FAT12 系统设计采用了如此短小的指针?
- 13.2 文件分配表组织不需要任何单独的机制来监测空闲空间。为什么?
- 13.3 在 Windows 操作系统的 NTFS 文件系统中, 一个文件的目录项中可能不包含指向相应文件的数据盘块的指针。为什么?
- 13.4 为什么 UNIX/Linux 操作系统的索引结点不包含文件名?
- 13.5 操作系统在什么时候挂载位于可移动磁盘驱动器上的文件系统?
- 13.6 为什么光盘的挂载不同于可移动磁盘驱动器?
- 13.7 当操作系统挂载了一个远程文件系统时, 远程文件系统是如何呈现在用户和应用程序面前的?
- 13.8 虚拟文件系统层曾经被用来支持访问远程文件系统。然而, 虚拟文件系统层拥有一个更为通用的目的。这一目的是什么?
- 13.9 简要描述内存映射文件比正常输入/输出更为高效的原因。
- 13.10 在 13.6 节, 我们曾提及某些实用例程命令并没有以实用程序的形式存在于系统上。这是为什么?
- 13.11 如果声称一个文件系统是事务型的或者是基于日志的, 这将意味着什么? 请就此进行简明扼要的解释说明。

296

磁盘调度和输入 / 输出管理

14.1 引言

在上一章中，我们从用户或应用程序设计人员的角度审视了输入与输出——操作系统给上层模块提供了哪些功能和服务？完成相关服务需要什么样的数据结构？有关功能是如何执行的？在这一章中，我们将会进一步去审查这些功能和服务在低层是如何实现的。特别地，任何文件系统的最底层都是由一组设备驱动程序和中断处理程序所构成的。在前面的章节中，我们曾经讨论了输入 / 输出功能是如何从简单的设备和结构起步而逐渐发展成为更加复杂的系统和服务。本章中，我们将会近距离地来观察和研究现代的硬件设备以及对它们进行有效而经济的管理所必需的操作系统组织方式。

[297]

在第 14.1 节中，我们将介绍低级输入 / 输出管理的问题，特别关注的是辅助存储器和磁盘驱动器。接着，在第 14.2 节，我们将会讨论输入 / 输出设备的一些大致分类以及彼此之间的区别。在第 14.3 节，我们则会描述一些用来支持输入 / 输出设备的通用的技术。然后，在第 14.4 节，我们将探索磁盘驱动器的物理结构，并在第 14.5 节讨论存储在磁盘驱动器上的信息的逻辑组织方式。在第 14.6 节中，我们将着重介绍廉价磁盘冗余阵列，其通过以特殊的构型对磁盘进行装配，从而能够获得更大的吞吐量和更高的可靠性。关于非常重要的磁盘调度操作及性能优化的话题，将会在第 14.7 节中展开讨论。在第 14.8 节中，我们将会介绍一种所谓内存直接存取（Direct Memory Access, DMA）控制器的特殊类型的设备控制器，其可以显著地减少由于输入 / 输出操作而带来的处理器负载。该节还会讨论影响操作系统行为的磁盘驱动器的一些特性。最后，在第 14.9 节，我们将对这一章的内容进行归纳总结。

14.2 设备特性

通过对输入 / 输出设备进行分门别类，可以笼统地把它们划分成操作系统区别处理的若干分组。我们在这里讨论其中的几种分类方法及相应的设备类别。

14.2.1 随机存取与顺序存取

在这一章，我们几乎只关注辅助存储设备，尤其是磁盘驱动器。磁带曾经一度被用作大型计算机中的辅助存储器。当个人计算机最初被开发出来时，它们也经常把磁带驱动器作为唯一的辅助存储设备——具体为原先开发用来保存音频的四分之一英寸规格的盒式磁带驱动器。不过，磁带有一项令人遗憾的特征，即它们不能支持数据的随机读取。为了到达和获取磁带上任意一个特定的数据片段，都必须穿越从磁头当前所在位置到所访问数据位置之间的所有的其他数据。即便是在速度非常快的磁带驱动器上，这也可能需要花上好几分钟的时间。幸运的是，磁盘驱动器没有此类特征。也正是由于存在这个不同，我们才提及磁盘驱动器是“随机存取”设备。尽管如此，当后面我们对磁盘驱动器展开详细介绍的时候，我们就会明白，这并不意味着它存取数据所耗费的时间是独立于数据的存储位置的。这一术语仅仅

是反映了把磁盘驱动器作为辅助存储器和采用磁带驱动器作为主要的辅助存储设备形成的对照中所凸现出来的差异。

14.2.2 设备分类

大多数的操作系统都将设备大致地划分为三种类别：块设备、字符设备和网络设备。其中每一类设备都拥有与其他设备存在根本区别的特征，而且每一类设备都可以按照一种有意义的方式来进行抽象。表 14-1 给出了关于这些设备类别的一些信息。

表 14-1 Linux 设备类别的特征

	块 设 备	字 符 设 备	网 络 设 备
随机存取	是	否	否
逆向查找	是	否	否
传输单位	盘块 (× 512 字节)	字符	数据包
软件	文件系统 / 裸设备	设备	协议

298

块设备

对于块设备 (block device)，每次会读取或写入一个盘块的数据 (一组字节，通常是 512 字节的倍数)。例如，各式各样的磁盘驱动器和磁带驱动器就是这样的设备。其间，盘块的大小在一定程度上是由硬件来决定的，因为磁盘控制器只能够读取或写入完整的磁盘扇区，但是，在设置文件系统的时候，盘块的大小也可由系统管理员来确定。一般情况下，盘块大小是物理扇区大小的某个小的倍数——最典型的为 4KB 或 8KB。这些设备往往支持对它们上面的任何盘块进行直接的随机存取，也就是说，可以按照任何次序来读取或写入盘块内容。文件系统通常位于块设备上，而且是用于访问这些设备的常规机制。对于随机访问块结构的设备而言，常常设立有高速缓存机制。同时，就像后面说明的那样，针对块设备的顺序存取往往会采用双缓冲技术。少数情况下，一些软件需要直接访问这些块设备，而不是通过使用文件系统来进行访问，这就是所谓的原始输入 / 输出 (Raw I/O)。此类软件的实例包括用于维护或检查文件系统自身的实用程序 (譬如，Linux 和 UNIX 系统的 fsck)，以及对辅助存储器有特别要求、足够精致和包含有高速缓存或磁盘调度操作的首选机制的软件 (例如，要求近似于苛刻的数据库服务器)。

字符设备

字符模式设备 (character mode device)，或称为字符设备 (character device)，每次仅传输一字节的数据。这样的设备包括打印机、键盘、鼠标 (以及其他的指针式设备)，等等。它们支持与块模式文件相同类型的大多数的基本操作，包括打开、关闭、读、写等。而为了执行一个不适合文件系统模型语义的操作 (例如，读取打印机的状态)，程序可以使用 ioctl 系统调用。显然，字符模式设备无法支持逆向查找。例如，我们不能读取从键盘上输入的 20 个字符之前的那个字符，也无法读取打印在前一页上的字符。不过，一些字符设备允许向前跳过若干字符。但是，字符模式设备从来不会采用高速缓存机制，即使它们可能拥有一个缓冲区。

网络设备

网络设备 (network device) 完全不适合采用传统的文件操作语义。其问题在于，等待源自网络输入的应用程序，永远不会知道什么时候数据才能输入和准备好，甚至都无法知道是

否存在可用的数据。一家公司可能会创建一个网站，并对通过该网站来销售各种的小型应用程序抱有很高的期望，但是其网站却可能从来没有接收到哪怕一次点击。出于这个原因，网络设备拥有一套完全不同于块设备和字符设备的读写操作的接口。

14.3 输入/输出技术

一般来说，输入/输出系统有两种工作方式。大多数的大型系统往往会拥有很多功能在或多或少地同时运行着，为此，能够有效处理这些功能的唯一的途径就是采用第2章中所描述的中断系统。与此同时，另一种所谓轮询（polling）的方法则常常用在装有低功耗类型的处理器的小型系统中。在轮询系统中，操作系统对设备的控制被编写在唯一的一个大的循环结构体里，其间，操作系统将依次检查每台设备的状态，并查看确认对应设备是否需要处理。这项技术往往用在嵌入式设备或简单的手持式游戏系统上，它们通常只有少量的可用设备，依次检查相关设备要比设置中断体系结构简单，而且还能够避免承受涉及中断服务的上下文切换所带来的开销。

输入/输出系统中的通用技术

在输入/输出系统中，常常会采用若干通用技术。在深入研究输入/输出系统的其他细节之前，我们首先来了解一下其中的一些通用技术。

14.3.1 缓冲技术

当我们向计算机系统输入数据的时候，我们通常是从一台设备读出数据，然后再写入另一台设备中。例如，当一个用户正在通过键盘输入方式编写一个文档时，计算机其实是把文档写入磁盘中。另一个极端的例子是，我们可能会将我们的硬盘内容备份到一台磁带驱动器上。上述任何情况，我们都会用到一种所谓缓冲（buffering）的技术。缓冲区是内存的一部分，其中存放了我们即将在输入/输出操作中使用的记录。关于我们为什么要使用缓冲区，存在诸多的理由支持。首先可能是因为要传输的数据的大小。用户编写文档时，每次只会产生唯一的一个字符，然而，我们却不能往磁盘中只写入一个字符。对于磁盘存取数据的最小单位是扇区。像磁盘和磁带这样的块设备只能以大的盘块来进行数据的传输。因此，我们要用一个缓冲区来暂存用户输入的那些字符，直到我们拥有足够的字符可以填满一个扇区时，我们再把该扇区的数据写入磁盘，然后开始另外一个新的扇区的存取操作。

在这种特定的场景中，磁盘的存取速度可能足以快到使我们在下一个按键可能到达之前，就能够把缓冲区中的全部数据写到磁盘上，从而腾空缓冲区使其可以接收下一个按键信息。但是，假设设备之间的速度差别很小——比如说3或4倍，在这种情况下，我们就可能需要采取一种稍微不同的技术，即双缓冲（double buffering）技术。我们将会为相应的进程分配两个缓冲区。我们首先应填满其中的一个缓冲区，然后开始把此缓冲区中的内容写到输出设备上。而在我们开始写设备操作的同时，我们就可以开始使用第二个缓冲区来存放输入的数据。当第二个缓冲区填满的时候，第一个缓冲区的写设备操作就应该完成了，于是，我们就可以开始将第二个缓冲区的数据写到输出设备上，同时开始再次把输入数据填入第一个缓冲区中。图14-1就显示说明了这个过程。在图14-1a中，我们可以看到进程A正在向1号缓冲区填入数据，而2号缓冲区正在等待。在图14-1b中，进程A已经填满了1号缓冲区，因此进程A就向2号缓冲区填入数据，同时把1号缓冲区中的数据写入磁盘驱动器上。

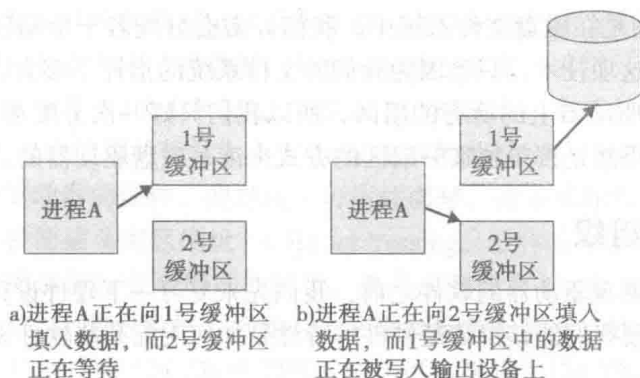


图 14-1 双缓冲技术应用示例

我们使用缓冲区的另一个原因可能是, 我们正在处理的两台设备都是块存取设备, 但是这两台设备却拥有不同的数据块大小。例如, 以太网适配器通常传输的最大数据块的大小大约是 1500 字节, 而令牌环适配器则允许传输最大约为 18 000 字节的数据块。如果要将一个数据包从令牌环连接传输到以太网连接, 我们就不得不使用缓冲区来保存令牌环数据包, 同时这个令牌环数据包拆分成多个以太网数据包然后再发送出去。(相关活动还牵涉其他复杂情况, 不过它们已经超出了本书的范围, 所以我们就不展开了。)

14.3.2 高速缓存技术

高速缓存 (caching) 是计算机系统中最具深远意义的技术之一。该项技术在无论硬件还是软件中都得到了运用。其目的是使一个容量较大、速度较慢但却比较便宜的内存, 看起来能够以与容量较小、速度较快因而比较昂贵的内存相同的速度运行。就像第 2 章和第 11 章中所讨论的那样, 高速缓存之所以能够发挥作用, 是因为进程实际上并不会随机地访问内存。相反, 进程是基于引用的局部性原理来运作的。这一原理表明, 与不接近那些已经引用过的地址的内存地址相比, 一个进程更可能去引用靠近那些已经引用过的地址的内存地址。例如, 一个进程在大部分时间里都是按顺序来执行指令的, 而不是以分支跳转的方式随机执行指令的。是的, 常常会有子程序的调用, 但一般需要许多指令来准备和构建下一次的子程序调用。同时, 进程往往会通过扇区、数组、字符串、数据包等来执行线性搜索。局部性原理的另一方面是说, 一旦一个进程引用了某内存单元, 那么该内存单元被对应进程再次引用的可能性要远远大于另外一个随机的内存单元被对应进程引用的可能性。再者, 通常情况下, 一个进程可能会花上一段时间来初始化一个表, 或者访问该表中的许多字段。

14.3.3 针对短小记录的分块技术

最后一种用在输入 / 输出系统的通用技术是分块 (blocking) 技术。分块就是将若干条逻辑记录打包成一个物理块写到一台设备上, 它有点类似于不同块大小的设备之间的缓冲技术。让我们来考虑原先设计是使用穿孔卡片但却被转换为运行在磁带上上的一个系统。记录编排可能都很接近于穿孔卡片的大小, 即 80 个字符。当然, 往一台磁带驱动器上写 80 字节的记录是可能的, 但是其效率却不会很高。各磁带记录之间有一个间隙, 可为磁带驱动器留出相应的时间, 用于把磁带驱动到合适的速度进而把磁带停在记录之间。这个间隙可以容纳许多大小为 80 个字符的记录, 故而对磁带造成大量的浪费。只是通过把 10 条记录打包成一个数据块, 并通过一次操作把该数据块写到磁带上, 我们就会节省出相当可观的空间。术语

300
301

“分块”的类似的运用是在磁盘文件系统中，我们经常会分配若干个扇区作为一个单独的数据块。之所以要采用这项技术，只是因为我们的文件系统的指针不够大，故而无法寻址到某些新型的大容量磁盘驱动器上的所有的扇区，所以我们只好一次分配多个扇区。尽管如此，在正常情况下，我们仍然是按单独唯一扇区的方式来读写磁盘驱动器的。

14.4 磁盘物理组织

在我们讨论控制磁盘驱动器的软件之前，我们先来复习一下硬件设计的相关内容，特别是磁盘物理组织，以便我们能够搞清楚硬件的特性是如何支配某些软件设计的。

14.4.1 扇区、磁道、柱面及磁头

在第3章中，我们在图3-2中展示了一张软盘。硬盘也是类似的，在图14-2中，我们揭示了一些其他的概念。在图中，我们可以看到两个盘片（platter）堆放在同一根主轴（spindle）上，从而使它们可以一起旋转。有4个传动手臂（或称之为磁臂）伸到了这两个盘片上，且每一个传动手臂包含有一个读写磁头。传动手臂可以沿盘片半径方向在盘片上移进和移出。当传动手臂在任何给定位置固定好之后，盘片就会旋转，这样磁盘表面的一圈就会从磁头下方通过，这一圈被称为一个磁道（track）。4个传动手臂是连在一起的，因而它们就可以作为一个部件一起移进和移出了。这就意味着在无须移动传动手臂的情况下，磁盘驱动器可以同时读取4个磁道的数据，其中每个磁道对应一个磁头和一个盘面。这组磁道被称作一个柱面（cylinder）。当然，如果有更多的盘片和更多的磁头，那么在一个柱面上就会有许多的磁道。一摞16个盘片堆叠在同一根主轴上，大概算得上是在现代驱动器上发现的最多的盘片数。

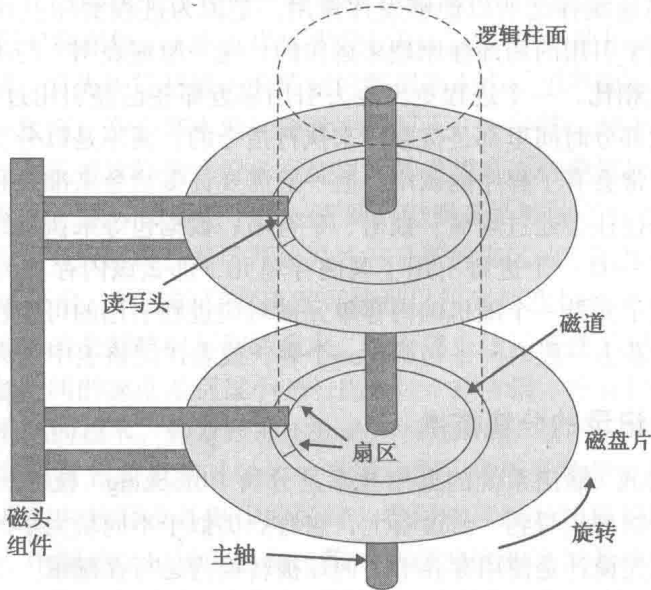


图 14-2 硬盘

一个磁道从逻辑上可以划分为若干个扇区（sector）。正如我们在前面所提到的那样，磁盘驱动器是块设备，所以它只能传输一个完整的数据单位，而不是按单个的字符来进行传输的。扇区是磁盘驱动器所能传输的最小的数据单位。在几乎所有的情况下，对于一个给定

302

的磁盘驱动器来说, 驱动器上的所有扇区的大小都应是相同的。在大多数现代的硬盘驱动器上, 这些扇区都分别包含 512 字节的数据, 外加一些附加信息。不过, 也有一些其他可用的扇区大小, 特别是 256 字节、1024 字节或 2048 字节。扇区大小几乎总是 2 的偶数次幂。采用 ISO 9660 标准的光盘, 使用的就是大小为 2048 字节的数据块。

这样来装配磁盘驱动器的硬件, 便导致了可由柱面号、磁头号 and 扇区号来指定的磁盘地址的概念, 或称之为柱面磁头扇区编址 (CHS addressing, 或称为 CHS 编址)。拥有 C 个柱面、H 个磁头且每个磁道包含 S 个扇区的磁盘, 总共应包含有 $C \times H \times S$ 个扇区, 通常情况下可存放 $C \times H \times S \times 512$ 字节的数据。例如, 如果磁盘标签上写着 C/H/S=4 092/16/63, 那么该磁盘就会有 $4092 \times 16 \times 63 = 4\,124\,736$ 个扇区, 故而可以存放 $4\,124\,736 \times 512 = 2\,111\,864\,832$ 字节 (2.11GB) 的数据。

14.4.2 扇区分组区和扇区编址

一个磁道上能够存储的二进制位的数量与磁头通过的线性距离是成正比的。硬盘驱动器以恒定的速度进行旋转, 而外层磁道的周长要比内层磁道的周长长, 因此, 按理说在外层磁道上能够存储比在内层磁道上更多的信息。但是, 旧式的磁盘驱动器对所有的磁道都采用了相同的定时 (译者注: 由此导致了所有磁道上存储的信息量都是一样的), 所以在外层磁道上的二进制位要比在内层磁道上的二进制位长些, 这可能是对二进制位的极大浪费。伴随时间的推移和技术的进步, 磁盘驱动器的电子器件已变得越来越复杂, 现在的驱动器甚至还包含有一个独立的计算机。为此, 自 20 世纪 90 年代中期以来生产的大多数的磁盘驱动器都采用了一种迥然不同的称之为区位记录 (Zone Bit Recording, ZBR) 的定时技术。其间, 磁盘磁道被划分成具有同样大小的磁道区 (或称之为扇区分组区), 并要求在各磁道区的磁道之间跨越时调整定时 (译者注: 不同的磁道区设立不同的定时)。这样, 外层磁道区的磁道上就可以设立更多的扇区, 而内层磁道区的磁道上设立的扇区数就要相对少一些。

鉴于各磁道上的扇区数对于整个驱动器来说不再是常量, 采用 CHS 格式编址扇区的想法也就不再能够正常发挥效用了, 所以某些东西必须要进行修正。但是因为某些软件在很大程度上是面向 CHS 概念的, 所以期望尽量保持或尽可能地接近相关格式。CHS 格式的磁盘地址共 24 位, 具体按如下方式划分成三部分:

柱面号	0~1023	(10 位)
磁头号	0~254	(8 位)
扇区号	1~63	(6 位)

因此, 可以用 CHS 编址方式表示的最大的磁盘地址是 8 GiB (译者注: 准确地讲, GiB 表示 2^{30} 字节, 而 GB 表示 10^9 字节)。为了遵循旧式驱动器的接口设计模式, 新式的驱动器继续采用了 CHS 编址方式, 同时需要告知操作系统其拥有一些非常大的柱面数、磁头数和扇区数。驱动器将会获取到相关参数, 并计算出一个逻辑块地址 (Logical Block Address, LBA)。这仅仅意味着磁盘的扇区是从 0 开始顺序编号的, 而且每个扇区都通过其 LBA 编号来进行标识。在此基础上, 驱动器将根据各磁道区中的相应的扇区数进一步计算目标块的实际物理地址。为支持操作系统无须在这种人为的 CHS 格式上耗费精力和直接传入 LBA 地址, 已经开发出了相应的基本输入 / 输出系统 (BIOS) 例程和驱动器。

然而到后来, 驱动器容量变得大到连能够指定的最大的 LBA 地址都不足以寻址到整个的磁盘驱动器了。为了能够容纳和适应这些更大的驱动器, 于是提出了新的地址格式以支持

28 位或 48 位的地址的使用。这样，假定每个扇区为标准的 512 字节，则磁盘大小的上限可以达到 128 GiB 或 128 PiB（译者注：PiB 表示 2^{50} 字节）。

14.4.3 低级格式化

当磁盘驱动器最初生产出来的时候，它们并不包含任何的信息，就连我们想要访问的扇区也不存在。这时候，必须要调用一种特殊的写操作模式，让磁盘把定义了扇区位置的字节数据真正地写到磁盘上。这种特殊的写操作模式就称为低级格式化（low-level formatting）。于是，每一个扇区都将会包含一个用来标识该扇区的柱面号、磁头号及扇区号的所谓扇区首部。一开始扇区是空的，其中没有信息，同时会附有一个校验和。（关于校验和的更多信息将在 14.5.3 节中进行介绍。）对于旧式的驱动器技术，会要求用户来完成此类的低级格式化。但值得庆幸的是，现在磁盘的低级格式化都是由制造商来完成的。

14.4.4 速度：寻道、传输及缓冲

硬盘的寻道时间（seek time）是操作系统性能的最重要的因素之一。它是指磁盘驱动器把磁头组件从一个磁道（或柱面）移动到另一个磁道（或柱面）所花费的时间。在大多数现代的系统，处理器在大部分时间里都是空闲的，其需要等待磁盘驱动器为其传输其所需要的信息。存取信息所用时间最大的影响因素在于要让读写磁头实际移动和放置到扇区对应的位置上。寻道时间的度量有若干种可能的方法。我们感兴趣的度量方法是平均寻道时间，但是为简便起见，我们就称之为寻道时间，因为平均寻道时间已经成为用来说明磁盘驱动器性能的一个行业标准了。抛开这个领域的一些早期的发展来看，磁盘驱动器的寻道时间的变化非常小。每年的变化率大约为 -8% 左右，照这样计算，10 年总共降低了 50%。在过去的 30 年里，这一判断还是相当准确的。现如今，一般的消费者的驱动器的平均寻道时间大约是 6~12ms，而最高性能的驱动器的平均寻道时间则大约是其一半左右，即降低到大约 3ms。

旋转延迟时间是与操作系统性能相关的因素之一。假设我们正在查找一个特定的扇区，于是我们首先会查找正确的磁道。当磁头组件到达对应的磁道的时候，它就会停下来。一个磁道上有好多扇区，而我们只是查找其中特定的那一个扇区。（我们可能要传输若干扇区的数据，但是我们将会指定从某一特定编号的扇区开始数据传输。）大多数情况下，下一个即将要通过磁头的扇区并不是我们正在查找的那个扇区。平均起来，在一半的旋转时间里我们会在错误的地方，这种延迟时间就称为旋转延迟（rotational latency）时间。旋转延迟时间随旋转速度成反比变化。表 14-2 给出了磁盘驱动器的最常见的旋转速度以及与之相对应的平均旋转延迟时间。

表 14-2 旋转延迟时间随驱动器旋转速度的变化

主轴转速 (RPM (每分钟转数))	平均旋转 延迟时间 (ms)	主轴转速 (RPM (每分钟转数))	平均旋转 延迟时间 (ms)
3 600	8.3	5 400	5.6
4 200	7.1	7 200	4.2
4 500	6.7	10 000	3.0
4 900	6.1	12 000	2.5
5 200	5.8	15 000	2.0

304

因为监控旋转位置对于操作系统来说并不是件非常容易的事情，而且寻道时间是如此之大以至于旋转延迟时间相比之下就不像是个重要的影响因素，所以直到最近之前，旋转延迟时间在很大程度上一直处于被忽略的境地。但是，要注意，现在旋转延迟时间已经摆到了和寻道时间几乎同等重要的地位上。因此，旋转延迟时间已开始 在磁盘调度算法中受到重视和给予相应的考虑。在本章后面的部分，我们将会就此展开进一步的阐述。

14.5 磁盘逻辑组织

通常情况下，应用程序把辅助存储器看作一组装满记录的文件。而在最底层，输入 / 输出系统则将磁盘驱动器看成是大量的扇区。为此，有必要对磁盘驱动器上的信息进行一些最基础的组织，以便使输入 / 输出系统能够找到其所需要的信息。因为个人计算机非常普及，所以在此我们主要描述个人计算机上的磁盘驱动器的组织情况。其他计算机平台会使用不同的组织方式，但是它们往往会拥有相似的元素。

14.5.1 分区

当 IBM 公司推出他们的第一台个人计算机的时候，那台计算机甚至都没有硬盘选项——软盘是其唯一的磁盘介质。当有了第一批可用的硬盘的时候，它们只能容纳 10MB 左右的信息，而且使用的是我们曾经在上一章讨论过的一种称之为 FAT12 的文件系统组织结构。没用多长时间，人们就认识到这种文件系统不会支持那些新的很快能够买得到的驱动器了。我们在前面就曾讨论过基于一次分配多个扇区的想法作为此类问题的一种解决方案。另外一种简单的解决方案是允许将唯一的一个磁盘驱动器划分成多个部分，同时将每个部分都看作是一个独立的驱动器，这样，旧式的文件系统仍然可以使用。这种解决方案称为分区[⊖] (partitioning)。由 DOS 系统提供的所谓 FDISK 的实用程序可用于把磁盘划分成若干独立的分区。最初版本的 FDISK 允许将一个驱动器划分为最多 4 个分区，且其中只能有一个分区可以包含一个可引导的操作系统。伴随 Windows NT 一同发行的新版的 FDISK，则允许在一个驱动器上定义更多的分区，并且允许有多个引导分区（也称作主分区）。这是向前迈出的很大的一步，尽管对于那些在这种格式改变之前就已创建的旧式的操作系统来说，会存在不兼容的问题。大多数现代的操作系统都支持这种格式的扩展分区表。因为这类实用程序并不经常使用，而且也没有太多的途径来进行功能的增强，所以曾经有一段时间，大多数其他的操作系统都只是使用了相同的实用程序。现如今，大多数的操作系统都提供了它们自己的分区实用程序，并且许多分区实用程序都拥有图形化用户界面。

事实证明，创建分区对于其他方面来说也是一项很有用的技术。首先，这是允许一台机器包含两种不同操作系统的一种简单直接的方法，并且还允许每个操作系统都认为自己可以唯一地控制磁盘驱动器。在图 14-3 中，我们可以看到，一个磁盘驱动器被划分成了包含有 3 种不同类型操作系统的 4 个分区。正常情况下，每个分区都会包含一个文件系统。但是，分区的另外一种用途是服务于那些非常专业的应用程序，这些应用程序想要管理它



图 14-3 包含若干不同的分区及操作系统的磁盘驱动器示例

305

⊖ 这项技术并不是起源于个人计算机系统。该技术在更早时候便由于这里提及的一些相同的原因而已经用在了某些大型主机系统上。

们自己的输入/输出而不是利用操作系统提供的缺省的面向文件的输入/输出。数据库管理系统可能就是这样的一个例子。此类系统针对它们期望看到的特定的存取模式做了大量的优化，因此，如果只能使用标准的操作系统的文件输入/输出，效率就不会很高。这种应用程序接口被称为原始输入/输出，其允许应用程序把分区看成是可以被随机存取的一个盘块数组，而不是通过普通的文件抽象机制来进行访问的。

但是到了后来，终于开发出来了新的拥有更大指针和可以支持特大硬盘的文件系统。一些应用程序需要比当时可以买得到的单个硬盘的容量还要大的文件空间，于是，分区机制又进行了逆向的处理应用，允许两个或多个磁盘通过分区机制组合到一起，从而在操作系统上层呈现为一个单独的驱动器。图 14-4 给出了跨越两个磁盘驱动器的单个分区的示例。

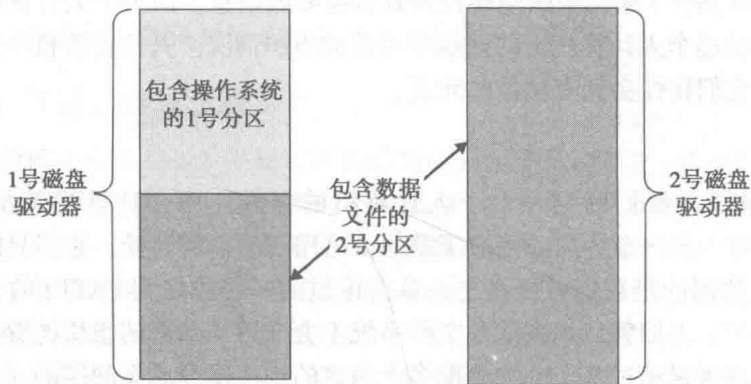


图 14-4 跨越两个磁盘驱动器的单一分区

306

14.5.2 引导块

在第 3 章中，我们曾经讨论过从磁盘驱动器启动系统的概念。当一台计算机重新启动的时候，其通常会试图从系统上的一个或多个设备来启动操作系统。大多数的个人计算机都包含有一块由电池供电的特殊的存储器，通常指的是互补金属氧化物半导体存储器（CMOS memory），有时也指基本输入/输出系统（BIOS）。在这个存储器的设置中，除了别的事项之外，将可以指定允许系统从中启动的一组不同的设备，且系统将按照指定的顺序来依次尝试从这些设备加以引导和启动。当系统试图从给定的某个设备启动时，对应设备也可能无法完成启动工作。例如，一个软盘驱动器或一个光盘驱动器里可能没有软盘或光盘，或者一个硬盘驱动器可能没有安装过操作系统。如果操作系统不能从某个设备启动，那么它将会试图从下一个设备启动。如果所有这些指定的设备都不能启动操作系统，那么一条并非普通用户就能明白的诊断错误就会显示在计算机屏幕上。相反，如果发现了能够启动操作系统的设备，那么在计算机硬件只读存储器（ROM）中的自启动程序（或称为自举程序）就会从该设备加载第一个扇区，并开始执行包含在此扇区中的代码。

无论分区怎样建立，关于硬盘分区的信息都会存储在磁盘上的第一个物理扇区中的某个部分。这个扇区称为磁盘的主引导记录（Master Boot Record, MBR）或者引导块（boot block），其中包含有分区表（partition table）。同时，引导块中还包含了一个小程序，该程序用来查看分区表，检查哪个分区是当前的活动分区（active partition），并进而读取对应分区的第一个扇区。对应分区的引导扇区中则包含有另外一个小程序，用来读取存储在该分区中的操作系统的第一部分（译者注：一般用来完成启动初始化工作）并加以启动执行。操作

系统的第一部分将会读取内核部分，并挂载相应分区中所发现的根目录结构。

当操作系统把自身引导装入内存之后，其将会挂载其在引导卷上所发现的文件系统。挂载文件系统的详情取决于操作系统及引导操作系统的分区的格式。另外，操作系统也可以挂载其他的分区。

14.5.3 错误检测与校正

当把信息写到一个硬盘单元里时，同时也会伴随着写入一些额外的信息。这些额外的信息是用来检测错误的，经常也会用来校正错误。有各种各样的用来创建和使用这些信息的方案，具体采用哪种方案取决于相应驱动器预计出现的错误类型、期望的可靠程度以及对驱动器的预期的相对价格。一般来说，越精致的技术往往会生产出越可靠的驱动器，但同时也会要求越复杂的计算，因而在磁盘驱动器中就会要求一个更快和更强大的处理器。此外，越复杂的技术往往会在磁盘上存储越多的冗余信息，从而导致相应磁盘会容纳越少的用户数据。因此，对于一个给定的用户存储的数据量而言，越复杂的技术将需要一个容量越大的驱动器。但是，错误校正技术支持制造商能够制造出速度更快、存储容量更大且在用户看来没有差错的驱动器。为了达到相同级别的可靠程度，存储数据的技术争取的越多，错误校正机制就需要变得越复杂。

在冗余信息的计算方面已经开展了大量的研究和开发工作，称之为差错检测码（Error Detection Code, EDC）或错误校验码（Error Correction Code, ECC）。所有这些技术都会计算关于数据块内容的一个函数，对应结果往往是一个很小的数。当把数据写入磁盘的时候，就会计算相应的函数值，并将计算结果和数据自身一起写到磁盘上。当从磁盘上读取数据的时候，会再次计算相应函数，并将计算结果同当初在数据写操作期间存储的计算结果值进行比较。如果这两个数值不一致，那么就说明读操作或写操作可能发生了一个错误。（读操作错误可能反映了从相应数据写入磁盘以来已经遭受了损坏。）

307

最简单的差错检测码包括循环冗余校验码（Cyclic Redundancy Check, CRC，或称为循环冗余校验），有时也称作纵向冗余校验码（Longitudinal Redundancy Check, LRC，或称为纵向冗余校验）。在这种方法里，所计算的函数被描述为一个多项式。例如，多项式 X^4+X^2+1 表示的就是二进制数 10101，此处的二进制数表示了多项式数值的指数的乘数。我们可以把 X^4+X^2+1 更确切地写成： $(1 \times X^4) + (0 \times X^3) + (1 \times X^2) + (0 \times X^1) + (1 \times X^0)$ 。计算过程可以被认为是用数据块中的数据去除以多项式所表示的二进制数，所得余数就是循环冗余校验码。这种类型的计算在计算领域，尤其是在网络方面，有着广泛的应用。因为网络中预计出现的错误类型与磁盘驱动器预料出现的错误类型有些不同，所以其所使用的多项式往往也是不一样的。表 14-3 中给出了几种常用的不同类型的多项式。有些时候，错误可能会发生在某一行的多个二进制数位上，这种情形称为突发错误（burst error）。一个多项式编码可以检测出任何一个长度小于或等于该多项式长度的突发错误。关于这种类型的计算，已经使用了有一段时间了，其主要原因于一套使用了移位寄存器的简单快捷的硬件的研发成功。

CRC-12 用于 6 位字符的串行通信线路，并会生成一个 12 位的循环冗余校验码。CRC-16 和 CRC-CCITT 均用于 8 位的串行通信中，并会生成一个 16 位的循环冗余校验码。后两种循环冗余校验码在美国和欧洲分别得到了普遍使用，并为大多数的应用程序提供了充分的保护。CRC-CCITT 用于磁盘驱动器上。CRC-32 会生成一个 32 位的循环冗余校验码。CRC-

32 多项式用于 IEEE-802 网络中，包括以太网、令牌环网以及无线局域网等。

表 14-3 几种常用的循环冗余校验码多项式

CRC-12	$X^{12}+X^{11}+X^3+X^2+X+1$
CRC-16	$X^{16}+X^{15}+X^2+1$
CRC-CCITT	$X^{16}+X^{12}+X^5+1$
CRC-32	$X^{32}+X^{26}+X^{23}+X^{22}+X^{16}+X^{12}+X^{11}+X^{10}+X^8+X^7+X^6+X^4+X^2+X+1$

在更现代的驱动器中使用了更为复杂的计算方法，具体包括海明码和里德 - 所罗门码。与各种各样的循环冗余校验码函数相比，它们会生成更多的冗余信息。一个典型的驱动器通常可能存储了 12 字节的冗余代码附着在 512 字节的数据块上，而且能够校正长达 22 位的突发错误。特别地，里德 - 所罗门码被应用于光盘驱动器中，于是，每帧中除存放了 24 字节的数据外，还存放了 8 字节的错误校验码用于错误校正的目的。之所以要求如此更高等级的冗余，是因为当一个系统处于运行过程中的时候，驱动器就会经常地被使用，比如一个汽车上的光盘播放器，而且此类存储介质很容易被损坏。

308

14.6 廉价磁盘冗余阵列

现代的硬盘是非常可靠的。我们一般根据平均故障间隔时间 (Mean Time Between Failures, MTBF) 来度量其可靠性。然而，我们可以通过以特殊的方式使用多个磁盘驱动器来进一步提高这种可靠性，并使其拥有更长的使用寿命。我们在这一节中所描述的技术称为廉价磁盘冗余阵列 (Redundant Arrays of Inexpensive Disk, RAID)。一些作者用“独立”(即“independent”)取代了“廉价”(即“inexpensive”)这个词，但是，后者才是在术语“RAID”首次被那些系统地研究由多个磁盘驱动器组成的阵列的使用的研究人员提出时，实际上所采用的那个词语。最初使用这个术语的目的是要表明，通过以巧妙的方式把廉价的磁盘驱动器组合到一起，就能够实现用较少的钱来获得很贵的磁盘驱动器才可拥有的可靠性的目标。不过，如今即便是廉价的磁盘驱动器也是非常可靠的。尽管如此，这些技术非常重要的一点就在于，为了让磁盘驱动器能够正常有效地工作，各个磁盘驱动器的故障模式必须是独立的。这便意味着磁盘驱动器必须运转在单独的输入 / 输出通道和输入 / 输出控制器上，从而获得最优的可靠性和性能。如果不能做到这一点，相关技术仍然可以确保数据不会丢失。当然，在替换共享部件的时候，数据有可能无法获取到。

对廉价磁盘冗余阵列构型的支持通常是在磁盘控制器中来完成的。但是，廉价磁盘冗余阵列不一定非得要有一个专用的控制器。对于廉价磁盘冗余阵列的某些构型来说，可能会通过操作系统中的一个软件模块来控制廉价磁盘冗余阵列的处理过程。现在，在大多数操作系统的软件模块里通常已提供了 0 级和 5 级的廉价磁盘冗余阵列构型 (即 RAID 0 和 RAID 5)。这些构型将会在 14.6.1 小节中展开进一步的介绍说明。

最初的廉价磁盘冗余阵列规格包括 6 种构型 (即 6 个级别)，分别称为 RAID 0、RAID 1、...，一直到 RAID 5。这些构型大多数都比使用一个单独的磁盘驱动器要更为可靠，其中的一些构型还在某些方面获得了性能的改善和提高，不过同时在其他方面的性能则有所下降和变差。RAID 0 是个例外，因为其仅仅提高了性能 (译者注：原书作者声称，RAID 0 仅仅在读操作方面提高了性能，应当属于笔误，因为其对读写操作均可并行展开，所以读写操作

方面的性能均应有所提高!),而对可靠性没有任何改变,甚至还有所降低。

14.6.1 廉价磁盘冗余阵列构型

下面即将给出的几幅图显示了廉价磁盘冗余阵列的若干构型,且每一幅图试图描绘的存储系统都持有相同数量的用户数据。在每种构型中的各磁盘驱动器的容量大小都是相同的,并且我们展现和说明了对于相应构型需要多少个磁盘驱动器就可以产生独立的 4 个磁盘驱动器的存储容量(译者注:其实,对于廉价磁盘冗余阵列来讲,数据信息真正存放在磁盘或者准确地说是存放在硬盘上,所以在廉价磁盘冗余阵列一节的其他部分,凡描述“磁盘驱动器”但强调其信息存储的,我们有时以“磁盘”或“硬盘”相称)。操作系统的高层部分将把每种廉价磁盘冗余阵列构型视为一个单独的磁盘驱动器,而其存储容量则是组成廉价磁盘冗余阵列构型的单独一个磁盘驱动器的 4 倍。廉价磁盘冗余阵列主要使用了三种技术,分别是磁盘镜像(把数据复制到一个以上的磁盘驱动器上)、条纹划分(将一个文件拆开和分别存放到一个以上的磁盘驱动器上)和错误校正(存储冗余数据,支持错误检测及可能的修复)。不同的廉价磁盘冗余阵列构型往往会使用这些技术中的一种或多种。

RAID 0——没有奇偶校验的条纹式磁盘阵列(striped disk array)。这种构型利用了数据条纹划分(data striping)技术,将每一个文件的数据块分散存放在多个磁盘驱动器中,不过,其没有提供冗余机制。尽管廉价磁盘冗余阵列技术的研发人员使用了术语条纹划分(strip)而非术语盘块(block),然而在实践中,相关实现总是基于数据块来处理 and 完成的。鉴于这种构型可以并行地执行多个读操作和多个写操作,所以在性能方面有所改善。不过,这种构型并没有增加容错能力(或称为容错性)。事实上,这种构型降低了可靠性,因为如果一个磁盘驱动器坏了,那么由于操作系统是把磁盘驱动器阵列看作了一个单独的磁盘驱动器,所以磁盘驱动器阵列中所有的数据都会丢失。如果有 N 个磁盘驱动器在运转,那将意味着,这种构型就会有 N 倍的出现故障的可能性。如果廉价磁盘冗余阵列的支持是由磁盘控制器来提供的,那么这时的操作系统就无法访问任何其余的磁盘驱动器。而如果相应的支持是由操作系统的设备驱动软件提供的,那么其余的磁盘驱动器在理论上依然存在被访问的可能性,不过,任何一个文件都不可能是完完整整地存放在其余的磁盘驱动器上的。如图 14-5 所示,磁盘中的数字表示将文件数据写入磁盘时的逻辑盘块号(在文件系统中可以看到)。

309

RAID 1——磁盘镜像(又称磁盘双工)。在 RAID 1 构型中,另有一套完全相同的磁盘驱动器(即磁盘的副本)。当往一块磁盘中写入任何数据的时候,该数据也会同时被写入该磁盘的副本(即磁盘镜像)中。如图 14-6 所示,其中带有阴影效果的那套磁盘就是磁盘镜像。假设一块主盘和它的镜像盘可以同时读取,那么这种构型提供的读事务的速率可以达到单个磁盘的两倍。这种构型对写事务的速率没有什么效果,因为每一数据块均须同时被写到主盘和镜像盘中,所以并行的效果也就丧失掉了。但如果有一个磁盘驱动器出现了故障,相关数据还将是很安全的,不过,这时性能会有所下降,因为在访问相应磁盘及其镜像盘的时候,二者只能有一个磁盘驱动器可用来对访问请求提供服务。这是代价最高的一种廉价磁盘冗余阵列构型。

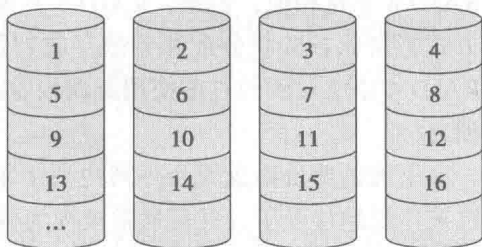


图 14-5 RAID 0——条纹式磁盘阵列

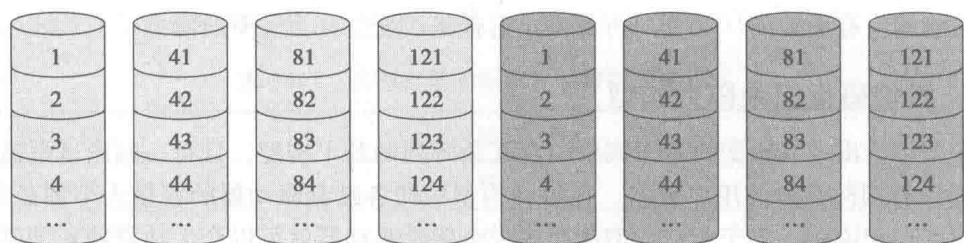


图 14-6 RAID 1——磁盘镜像

RAID 2——纠错编码 (error-correcting coding) 和 RAID 3——位交叉奇偶校验 (bit-interleaved parity)。这两种技术已被证明是代价高得惊人而且效果还不如其他技术，它们还要求磁盘旋转须保持同步以获取高性能。如今，它们已不再使用了。为此，在这里我们就不对它们展开详细描述了。

310

RAID 4——奇偶校验专用磁盘 (dedicated parity drive)。这种构型提供了数据块级别的条纹划分 (就像 RAID 0) 并设立有奇偶校验磁盘。其中，被写到奇偶校验专用磁盘的所谓奇偶校验块用来校验其所在条纹的其他的数块，在这个例子中，如“奇偶校验块 1~4”应涵盖数据块 1、2、3、4，以此类推，如图 14-7 所示。如果有一个数据磁盘出现了故障，那么相应的奇偶校验数据就会被用来创建一个替代磁盘。RAID 4 的一个劣势在于，每写一个数据块，就必须同时对相应的奇偶校验块进行读取、重新计算和重写操作。为此，奇偶校验磁盘就成为一个输入 / 输出瓶颈。这种构型提供了与 RAID 1 几乎同样的可靠性，但如果有一个磁盘驱动器坏了，对性能的冲击将是非常严重的。尽管如此，其代价只是唯一额外的磁盘驱动器，因此，如果廉价磁盘冗余阵列拥有多个磁盘驱动器，这种构型在价格方面还是非常有优势的。

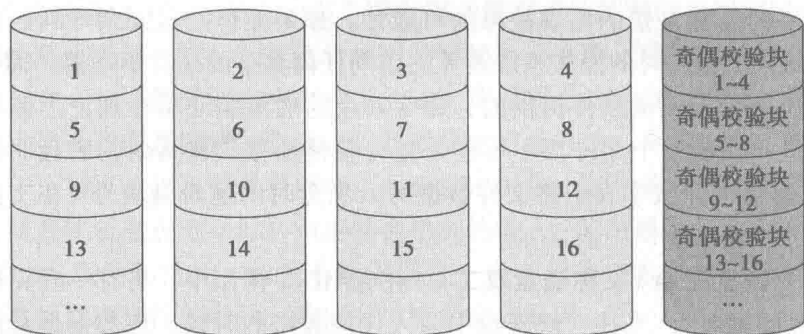


图 14-7 RAID 4——奇偶校验专用磁盘

RAID 5——块交叉分布式奇偶校验 (block interleaved distributed parity)。RAID 5 与 RAID 4 非常相似，但是，RAID 5 并没有把奇偶校验块全部放到同一块磁盘上，而是以轮转方式把奇偶校验块分配和分散存放到各块磁盘上，如图 14-8 所示。这项技术消除了我们在 RAID 4 中所看到的过度使用奇偶校验盘的问题。RAID 5 是最受欢迎的廉价磁盘冗余阵列构型之一。

下面的廉价磁盘冗余阵列构型并非最初的廉价磁盘冗余阵列规格的组成部分，但是它们已被广泛地接受了，并且通常被视为标准。

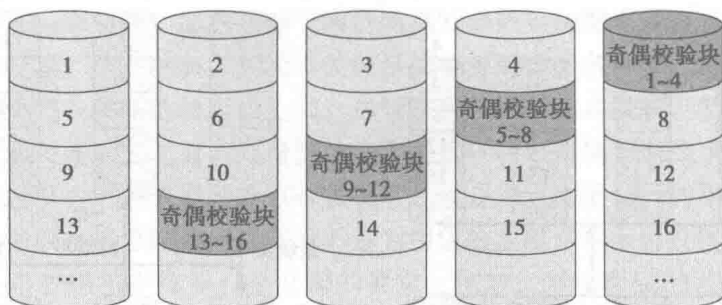


图 14-8 RAID 5——块交叉分布式奇偶校验

RAID 6——独立的双奇偶校验数据磁盘 (independent data disk with double parity)。本构型中, 提供了 RAID 5 中的数据块级别的条纹划分和奇偶校验数据跨多磁盘的分布式存放。但是, 其并没有仅仅采用一套简单的奇偶校验方案, 而是同时利用两种不同的算法来计算奇偶校验值。在实现 RAID 6 构型的过程中, 使用了包括双重编码校验 (具体为奇偶校验码和里德 - 所罗门码)、正交双奇偶校验、对角双奇偶校验等在内的多种计算方法。如图 14-9 所示, 其中, 对于两种不同的奇偶校验块, 分别在相应条纹中标以函数 P 和函数 Q 来加以区分, 同时还分别配上了对比鲜明的阴影。同 RAID 5 相比, RAID 6 需要多增加一个磁盘驱动器, 但其将可以承受同时损坏两个磁盘驱动器的故障情况。

311

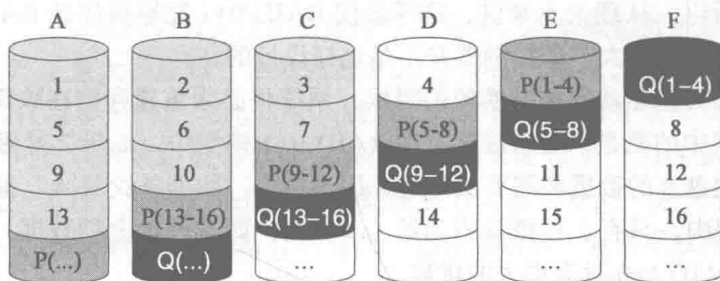


图 14-9 RAID 6——独立的双奇偶校验数据磁盘

RAID 0+1——条纹式单镜像 (mirror of stripes)。在这种构型中, 将创建两套 RAID 0 数据条纹 (即数据块), 并通过一套 RAID 1 镜像机制来具体实施, 如图 14-10 所示。其中, 条纹划分技术改进了性能, 而镜像技术则增强了可靠性。一般来说, 这种构型表现要优于 RAID 5。

RAID 1+0 (又称 RAID 10)——条纹式多镜像 (stripes of mirrors)。设立多套 RAID 1 镜像机制 (即所谓的磁盘驱动器镜像对, 或简称镜像对), 同时支持 RAID 0 条纹划分, 如图 14-11 所示。这种构型与 RAID 0+1 有着相似的性能和可靠性特征。但是, 当一个磁盘驱动器发生故障的时候, 其性能和可靠性会稍好一些。具体而言, 对于 RAID 0+1 构型而言, 一个磁盘驱动器的损坏意味着相应整个数据条纹集的丢失, 所以另外的一个数据条纹集必须承担所有的数据存取工作。而对于 RAID 1+0 构型来说, 只有对应损坏了磁盘驱动器的镜像对的另一个磁盘驱动器才须承担相应镜像对的所有数据存取工作, 而其他镜像对可以像以往一样正常、高效、可靠地工作。

14.6.2 廉价磁盘冗余阵列故障

在建立 RAID 0+1 构型时, 我们可以使用两个控制器来创建数据条纹集, 同时使用设备驱动程序级别的软件来实现数据条纹集的镜像。就像前面所描述的那样, 在这种情况下, 单

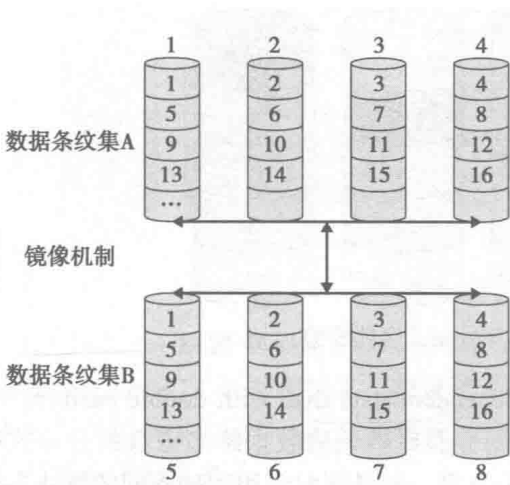


图 14-10 RAID 0+1——条纹式单镜像

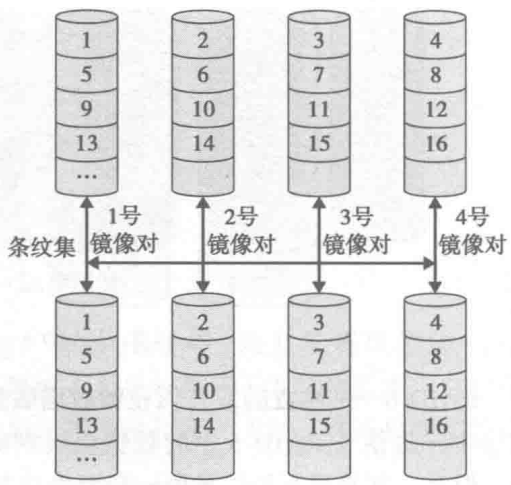


图 14-11 RAID 1+0——条纹式多镜像

个磁盘驱动器的故障会导致整个数据条纹集的丢失。但是，如果运行在 RAID 0+1 构型的控制器了解整个构型配置，那么当譬如说 3 号磁盘驱动器发生故障时，其将会继续对文件进行数据条纹划分并分别存储到数据条纹集 A 的其他 4 个磁盘驱动器上。而如果后来 6 号磁盘驱动器也发生了故障，其还可以使用 2 号磁盘驱动器作为替代进行处理，因为此二者应当拥有相同的数据。所以，从理论上讲，这样会使 RAID 0+1 能够提供像 RAID 1+0 那样的容错性。然而，遗憾的是，大多数控制器并不是这样设计的。

当出现故障的磁盘驱动器被替换掉的时候，系统将必须重建存储在故障磁盘驱动器（译者注：实际上是其中的磁盘）上的信息。在 RAID 0+1 构型中，如果 2 号磁盘驱动器发生了故障，那么 5 个硬盘上的数据将需要被重建，因为整个“数据条纹集 A”都将被消除掉。而在 RAID 1+0 构型中，只有 2 号磁盘驱动器（译者注：即其硬盘上的数据）必须重建。在这里再一次强调，RAID 1+0 具有更大的优势。

还有好多其他的有专利权的廉价磁盘冗余阵列构型，包括许多商标性的术语。这些构型在某种特定的情况下可能有某些好处，也可能没有某些好处。对测试构型的分析，尤其是对它们在各种故障情景中的表现的分析，并不是一件小事，但在特殊的情况下可能是担保性质的。

在比 RAID 0 更高级的 RAID 构型中，当有一块硬盘出现故障的时候，整个阵列可以继续运行，只是有时性能会比较低，而有时我们会面临发生更多故障的风险。例如，在 RAID 1 构型中的一块硬盘的损坏通常意味着，此刻我们将会面临某种增加的风险，因为一旦相应镜像对中的另一块硬盘也出现故障，我们就会丢失该镜像对上的所有的数据信息。为此，在只有一块硬盘出现故障的情况下，我们可以继续保持廉价磁盘冗余阵列的运行。这时候，我们会注意到在执行某些读操作的过程中性能有些下降，这是因为我们现在只有一块硬盘来完成这些读操作，而在那块硬盘损坏之前，我们是有两块硬盘来共同协作和共同承担完成相应的读操作任务的。在其他的某些场景中，我们也可能会因为系统的性能变得无法接受而不得不关闭系统。例如，在 RAID 6 构型中，如果我们坏了一块硬盘，写操作的性能将会变得非常糟糕，因为我们必须要读取所有其他的硬盘，才能够计算出故障硬盘的奇偶值，这已经到了能被接受的边缘线上。然而，只要坏掉的那块硬盘被替换之后，我们就可以开启重建存储在故障硬盘上的信息了。

这样便提出了一个问题，那就是对于廉价磁盘冗余阵列构型来说，我们希望使用那种可以“热插拔”的硬盘。这意味着，在无须关掉系统电源的前提下，就可以拔下出现故障的那块硬盘，同时把一块新的硬盘插上去。尽管这种技术很容易理解，但是它确实需要特殊的硬件，且相关硬件要比普通硬盘贵出许多。这项决定可能要取决于保持系统运行对于某些操作来说是否至关重要或者是否只有数据存在才是至关重要的。如果是后一种情况，我们可能会倾向于采取系统停机处理措施而不是为相应磁盘驱动器多付价钱。

而有些情况下，保持系统运行是必需的要求。例如，对于医院的系统来讲，其对病人的护理可能是非常关键的，系统故障可能就意味着生命的消逝。对于这样的案例，我们可能会选择更进一步的应急预案，即在架子上配备一个空置的磁盘驱动器时刻准备着，一旦发生某个磁盘驱动器出现故障的情况就立刻将其插到系统上。在极端情况下，我们可能会让一个磁盘驱动器处于暖备份（warm standby）状态——这个磁盘驱动器已经插入磁盘驱动器架上，只是尚未加电，或只是没有旋转。当操作系统检测到发生故障的时候，就会为该热备份磁盘驱动器加电和使其旋转起来，并开启相应数据重建过程。当然，磁盘镜像是热备份（hot standby）的极端形式。

14.7 磁盘操作调度

我们在前面曾经提到过，就系统性能而言，寻道时间是磁盘驱动器的最重要的度量指标之一。已经证实，如果我们有大量的磁盘操作要完成，那么我们处理相关请求的次序会对系统的整体性能产生重大的影响。在过去至少 20 多年的时间里，处理器的性能大约以每年 50% 的速率一直在稳步提升。与此同时，磁盘驱动器性能的提升速率仅为每年 10% 左右。为此，如果我们可以牺牲处理器的一些速率来改善磁盘系统的性能，应当是合乎情理的。为了说明这个道理，让我们来假设，我们现在有需要服务的一系列的磁盘请求，同时，磁盘驱动器的一个寻道操作将会把所有的磁头一起移动到某个磁道或柱面上。因此，我们将只需考虑磁道号，并应认识到我们实际上在同时定位（可能）许多磁头——当然，至少两个。所以，我们将会把那些由系统中运行的诸多进程提出而到达输入 / 输出系统的磁道号列成一个表，进而查看执行相关请求所必需的寻道时间，然后来确认我们是否可以就此有所改进。在所有这些情景中，我们均假设磁盘驱动器拥有 80 个磁道，磁头目前停留在 28 号磁道上，而且我们面临如下以队列方式表示的一组请求：

17, 30, 24, 37, 15, 27, 11, 75, 20, 5

314

14.7.1 先来先服务调度算法

处理这些请求的最简单的方法就是按照它们在队列中的先后顺序来依次为它们提供服务，称之为先进先出（First In First Out, FIFO）调度算法，又常称为先来先服务（First Come First Served, FCFS）调度算法。这种算法吸引人的地方在于其实现起来非常简单，同时还有一个优点就是，它是公平的。其公平性具体体现为，首先提出请求的进程首先获得服务。不过，这种算法不一定能够获得最好的系统总体性能。此外，我们以后将会看到，甚至对于单独的一个应用程序而言，这种算法也可能不会达到最佳的性能。在实施先来先服务调度算法的过程中，操作系统将会按照各请求在队列中的先后顺序在磁道间移动磁头。因此，磁头首先会从 28 号磁道移动到 17 号磁道，然后移动到 30 号磁道，接着再移动到 24 号磁道，依次类推。按照这种顺序处理该请求队列时，系统寻道将总共会移动磁头跨越 227 个磁道，如

图 14-12 所示。因为我们不太清楚期间所涉及的旋转延迟时间，所以我们将使用系统为了服务那些请求而寻道时需要移动磁头跨越的磁道数，作为我们衡量相关算法是否高效的度量标准。

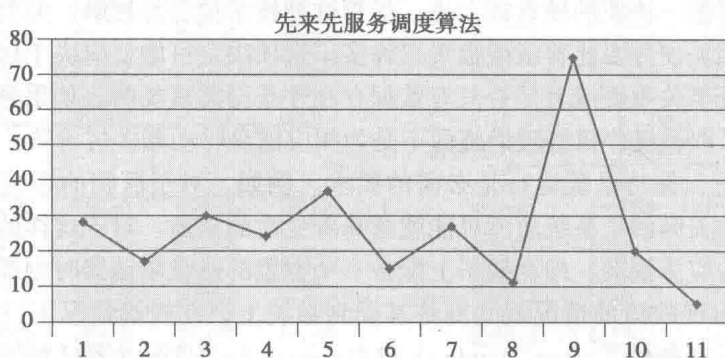


图 14-12 先来先服务调度算法

14.7.2 搭便车调度算法

搭便车 (pickup) 调度算法是被某些专家推崇和提起的关于先来先服务调度算法的一种变种算法。在这种算法里，服务请求通常采取与先来先服务调度算法相同的排序方法，但是在系统移动磁头的过程中，它将会在即将经过的有操作请求在队列中的任何一个磁道处停留下来并给予相应的操作服务。（相应调度实现模块在 Linux 系统中称为 Noop 调度器，这里 Noop 即 No Operation。）例如，对于我们的例子所给出的请求队列，磁头一开始是在 28 号磁道上，然后开始向队列中的第一个请求，即 17 号磁道移动。但在移动行进途中，其将会把 27 号、24 号以及 20 号磁道的请求捡拾起来先行予以服务处理。为此，实际调度服务所对应的整个序列应当是：

27, 24, 20, 17, 30, 37, 15, 11, 75, 5

上述调度过程最终导致的总的寻道时间即移动磁头所跨越的磁道数为 191 个，相对于先来先服务调度算法而言有了相当大的改进。图 14-13 具体说明了针对该示例的基于搭便车调度算法的调度过程。

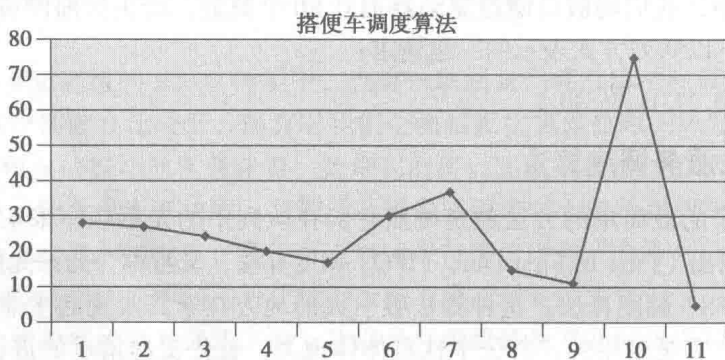


图 14-13 搭便车调度算法

14.7.3 最短寻道时间优先调度算法

接下来，我们来看一下一种所谓的最短寻道时间优先 (Shortest Seek Time First, SSTF) 调度算法，有时也称为最短定位时间优先 (Shortest Positioning Time First, SPTF) 调度算法。

当我们在操作系统的其他地方（如虚拟内存页面置换或进程调度）使用类似算法的时候，我们经常会说，诚然它们是最好的算法，但是我们却不能真正地利用它们，因为我们无法预知未来。然而，对于磁盘调度来讲，我们是可以真正利用这种算法的，因为我们所关注的所有请求都在我们所查看的队列中。再次地，一开始我们把磁头放在 28 号磁道上，在队列中离 28 号磁道最近的请求项是 27 号磁道，于是我们接下来把磁头移动到那个磁道。为此，现在的下一个最近的请求项需要返回到 30 号磁道，所以我们进一步把磁头移动到那里。整个过程如图 14-14 所示，相应调度服务所对应的整个序列是：

28, 27, 30, 24, 20, 37, 17, 15, 11, 5, 75

其间，移动磁头所跨越的磁道总数为 133 个。因此，这种算法的性能几乎是先来先服务调度算法的两倍。尽管如此，这种算法却是非常不公平的。队列中的第一个请求，竟然是其中等待时间最长的，直到队列中大约一半的请求都被调度过了，它才得到了服务。同时还要注意，磁头一直在磁盘的中间部分来来回回地移动，故而要访问位于磁盘中间磁道上的数据块的进程，往往会比要访问位于磁盘中心磁道或边缘磁道的数据块的进程能够得到更好的服务。另外请注意，在这种算法运行的同时，操作系统可能会产生新的更多的磁盘操作请求，并会和剩余未处理的请求一起按照最短时间优先调度的次序放置在队列中。因为要访问磁盘中间磁道的那些进程会受到更多的青睐和关照，所以它们也会有更多的机会冲到队列靠前位置插入另外的请求，从而进一步扩大了它们的优势。这种算法可以被视为是对某些请求赋予更高的优先权——尤其是最靠近磁头当前位置的那些请求。与任何优先级机制一样，我们必须关注那些较低优先级请求的饥饿现象。鉴于此，可以采取某种措施，使远离磁头当前位置的那些数据块的访问请求的优先权逐步得到提升，从而使它们尽快得到相应的服务。有时把最短寻道时间优先调度算法的这种变种算法称为请求时变式最短寻道时间优先（Aged Shortest Seek Time First, ASSTF）调度算法。这种算法主要需要调整实际的“寻道时间”（译者注：具体表征为从磁头当前所在磁道到达目标磁道所需跨越的磁道数），即将其减去“一个加权因子和相应请求在队列中的等待时间的乘积”。如果设 T_{eff} 表示一个请求的有效的（或加权的）寻道时间， T_{pos} 表示寻道所需的实际时间， W 表示我们想分配给过往请求的一个加权因子， T_{wait} 表示相应请求在队列中的等待时间，那么请求时变式最短寻道时间的计算公式将是：

$$T_{\text{eff}} = T_{\text{pos}} - W \times T_{\text{wait}}$$

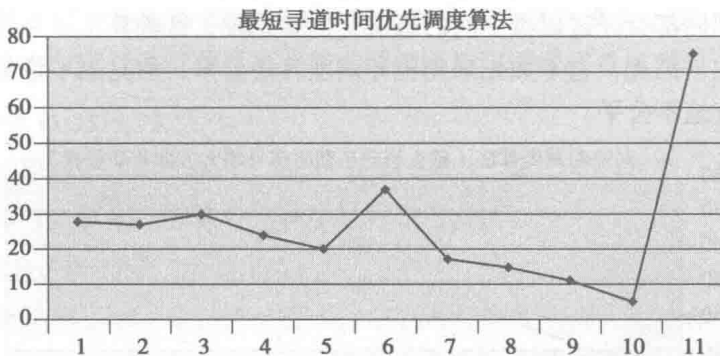


图 14-14 最短寻道时间优先调度算法

14.7.4 向前看调度算法

我们即将要学习的下一种算法通常称为向前看（LOOK）调度算法，它还有一个很流行

的名字，即电梯调度算法（elevator algorithm）。在该算法中，一旦操作系统按照某一个方向开始移动磁头和查找磁道，将一直沿着该方向向前推移，且直到在该方向上没有其他要访问的磁道时，才会逆转过来在反方向上开始查找。换句话说，系统会一直向前“搜寻和查看”，以决定什么时候逆转寻道方向。这类似于电梯的工作方式。一旦电梯开始上升，它就会只朝向上这个方向移动，直到它在这个方向上没有请求为止，然后它才会逆转过来下行移动。（就像许多类比一样，这种算法同真正的电梯工作相比还是会有点不一样的地方。因为电梯还会考虑来自某一层的请求是上行还是下行，而且如果它正在下降，它是不会为想要上升的用户停下来的。但是操作系统只需要把磁头定位到磁道即可，寻道请求并没有包含寻道方向的概念。）让我们再次来看看我们的磁道访问请求序列，我们同时还假设操作系统按照磁道号减小的方向开始移动磁头和查找磁道。在这种情况下，磁道请求调度服务的顺序将会是：

28, 27, 24, 20, 17, 15, 11, 5, 30, 37, 75

移动磁头所跨越的磁道总数为 93 个，如图 14-15 所示。

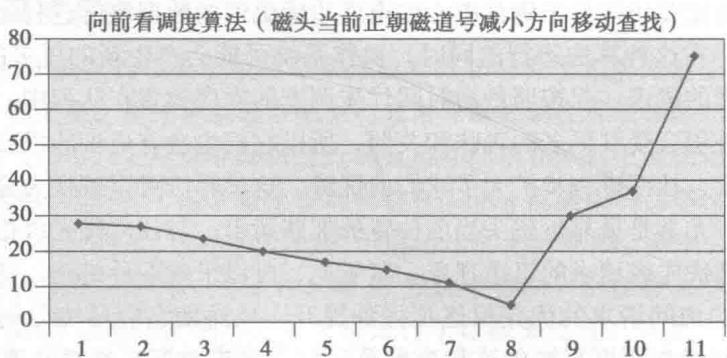


图 14-15 向前看调度算法（磁头当前正朝磁道号减小方向移动查找）

而如果操作系统按照磁道号增大的方向开始移动磁头和查找磁道，那么磁道请求调度服务的序列将会是：

28, 30, 37, 75, 27, 24, 20, 17, 15, 11, 5

移动磁头所跨越的磁道总数为 117 个，如图 14-16 所示。无论哪一种情形，向前看调度算法都比先来先服务调度算法或最短寻道时间优先调度算法要好。然而，每次无论磁头移向哪一端，其来来回回都会经过磁盘中间的磁道；也就是说，这种算法还是倾向于关照磁盘中间磁道上的数据块。因此，它不如先来先服务调度算法公平，但是也不像最短寻道时间优先调度算法那样极度地不公平。

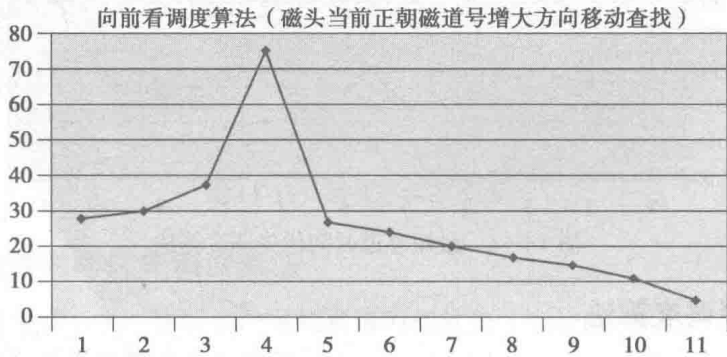


图 14-16 向前看调度算法（磁头当前正朝磁道号增大方向移动查找）

一些专家还讨论过一种另外的算法称为扫描算法 (SCAN)。这种算法和向前看调度算法是类似的, 但是当队列中没有在磁头当前行进方向上的磁道访问请求时, 该算法并不会改变磁头的移动方向, 而是沿着当前行进方向继续移动直到走完全程到达磁盘的另一端。因为没有人愿意真正去实现这种算法, 所以我们略过这种算法。这里之所以会提到这种算法也只是出于完备性的考量, 因为一些其他算法的名称与“扫描”有一定的相关性。

14.7.5 循环向前看调度算法

为了构造一种更公平的算法, 就在向前看调度算法的基础上设计了一种变种算法。当磁头移动到某一方向的最后一条磁道时, 其将不再逆转方向和查找最近的磁道, 而是在逆转方向后由操作系统控制去查找队列中在相反方向上最远的磁道访问请求。然后再回过头来, 按照之前查找磁道的方向开始执行寻道操作。其目的是要消除给予磁盘中间磁道上的文件的不公平的优越地位, 因为在其中一遍穿越磁盘中间的过程中, 并未对任何磁盘操作请求提供服务。这种算法称为循环向前看 (Circular-LOOK, C-LOOK) 调度算法, 有些专家也称之为圆筒状向前看 (cylindrical-LOOK) 调度算法或循环电梯 (cyclic elevator) 调度算法。该算法名称所基于的想法是, 把磁盘的地址空间 (即磁道号) 缠绕在一个圆柱体上, 于是查找完 0 号磁道后, 下一个要考虑查找的磁道就是 80 号磁道 (译者注: 此处实际上潜在假设了磁头查找方向为磁道号减小的方向。另外, 在 14.7 节开始时曾提到本节示例假设磁盘拥有 80 个磁道, 故而磁道号变化范围应为 0~79, 因此, 准确地来说, 此处的 80 号磁道应为 79 号磁道)。遗憾的是, 这种算法将导致在磁盘调度服务的中间过程会花费很长的寻道时间 (译者注: 指磁头沿查找方向到达尽头时逆转折返回来时的开销)。不过, 这个漫长的寻道过程也不会像它看上去那么糟糕。一般来说, 当我们讨论磁盘驱动器的寻道性能时, 我们会引用平均寻道时间。然而, 寻道时间并不会随着寻道距离而线性地增加。就像一个移动的物体, 磁盘上持有磁头的磁臂往往是缓慢地开启移动, 然后逐渐会变得越来越快。因此, 跨越整个磁盘的寻道所花费的时间不会是平均寻道距离所花费时间的两倍, 而是会比此要稍微快点。鉴于不可能对此做出准确的预测, 所以我们将不予理睬这些, 而只是简单地进行与之前我们所做的相同的计算, 保证情况不会真的有这么差就行了。再次说明, 确切的磁道跨越总数将会取决于我们开始移动的方向是磁道号减小还是磁道号变大的方向。由于在调度服务序列中部的磁道访问请求的较大的寻道时间开销, 磁头移动的这两个方向给循环向前看调度算法所带来的不同, 不会像它们对向前看调度算法曾经显现出来的区别那么厉害。如图 14-17 所示, 当沿着磁道号减小方向进行磁道查找时, 相应的磁道访问调度服务序列为:

28, 27, 24, 20, 17, 15, 11, 5, 75, 37, 30

磁头总共移动跨越了 138 个磁道。而在图 14-18 中, 我们可以看到, 当沿着磁道号增大方向进行磁道查找时, 相应的磁道访问调度服务序列为:

28, 30, 37, 75, 5, 11, 15, 17, 20, 24, 27

磁头总共移动跨越了 139 个磁道。循环向前看调度算法的诱人之处在于, 其提供了较低的服务可变性 (service variability) ——任何给定的磁盘请求的性能通常都具有更好的可预测性, 并且更少地依赖于文件在磁盘上的位置。只有当磁盘访问达到非常高的负载级别时, 循环向前看调度算法才会因为能够减轻饥饿问题而比向前看调度算法更有优势。就像前面所提到的扫描算法, 一些作者也曾经讨论过循环扫描算法 (C-SCAN), 该算法在逆转磁头移动

方向之前，也会要求磁头移动到到达行进方向的尽头，即磁盘的另一端。

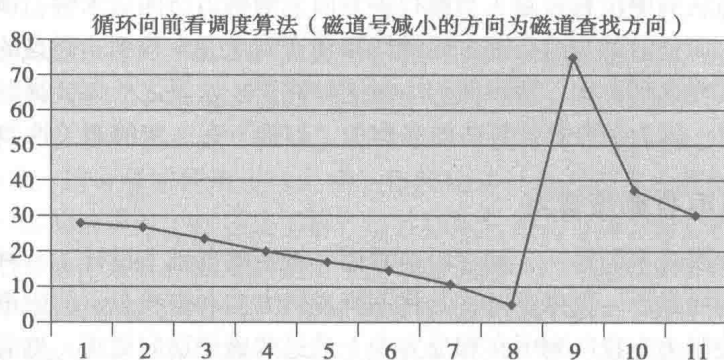


图 14-17 循环向前看调度算法（磁道号减小的方向为磁道查找方向）

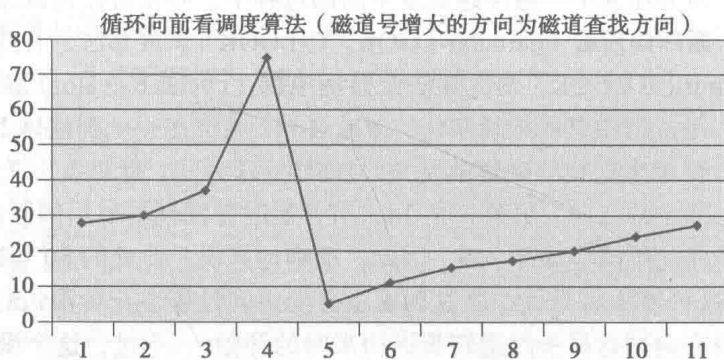


图 14-18 循环向前看调度算法（磁道号增大的方向为磁道查找方向）

14.7.6 先来先服务 – 向前看调度算法

另外一种磁盘调度算法通常称为先来先服务 – 向前看调度算法（译者注：本算法英文名称为 F-SCAN，直译为先来先服务 – 扫描算法，但其实本算法并非由扫描算法而是由向前看调度算法演化而来，所以这样确定中文名称更为恰当），是向前看调度算法的变种算法。这种算法使用了两个队列，不妨称作队列 X 和队列 Y。系统开启时首先从队列 X 开始扫描，且当启动扫描时，系统将会冻结请求队列 X。也就是说，在这轮扫描已经开始并进行着的过程中，新到达的任何一个请求都将会被放入队列 Y 中。当完成队列 X 的扫描之后，就会冻结队列 Y，然后开始新一轮的对队列 Y 的扫描，同时在此期间到达的任何一个请求都会被放置进入队列 X。这种方法避免了循环向前看调度算法高代价的漫长寻道开销，对先来先服务调度算法的公平性和向前看调度算法的高效性进行了很好的折中处理。因此，这种算法能够避免长时间的饥饿问题。

14.7.7 N 轮向前看调度算法

向前看调度算法的最后一种变种算法是按照 N 组请求来分批依次加以处理，先扫描完一批请求，之后再处理下一批请求，称之为 N 轮向前看调度算法（译者注：本算法英文名称为 N-step SCAN，直译为 N 步扫描算法，但其实本算法并非由扫描算法而是由向前看调度算法演化而来，所以这样确定中文名称更为恰当）。就像先来先服务 – 向前看调度算法一样，N 轮向前看调度算法可以防止无限期的延迟（饥饿）。N 轮向前看调度算法的另外一个目的是，

就一个没有被服务的请求可以等待多长时间设置一个上限。因此,对于负载非常重的系统和拥有大量软实时应用程序的系统来说,这种算法是非常有用的。注意, N 轮向前看调度算法的效果在很大程度上取决于 N 的大小。如果 N 等于1,那么 N 轮向前看调度算法实际上就是先来先服务调度算法,而如果 N 足够大,即大到几乎所有的请求在第一轮扫描中都可以得到服务,那么 N 轮向前看调度算法就相当于向前看调度算法。

14.7.8 Linux 调度程序

与大多数的操作系统相比, Linux 系统拥有更多可用的调度程序变种。在此,我们将主要研究一下其当前所支持的三种调度程序,它们均基于我们已经介绍过的各类算法而演化形成。

预测式调度程序

预测式调度程序 (anticipatory scheduler) 曾一度是 Linux 系统中默认采用的磁盘调度程序。其借鉴了搭便车调度算法的请求合并处理机制,同时又使用了类似于向前看调度算法的单程电梯序列。其独特之处在于,如果它认为一个进程会在附近请求存取更多的数据,那么在一个同步的读命令完成后,它会通过拖延一段时间来尝试预测读操作请求。而如果有一个新的请求是来自于最后一个进程且与当前位置相距不是太远,那么它就会逆转寻道方向。

[320]

基于截止时间的调度程序

基于截止时间的调度程序 (deadline scheduler) 也综合运用了搭便车调度算法的请求合并处理机制及类似于向前看调度算法的单程电梯序列,并为所有的操作强加了一个截止时间,用以预防资源饥饿问题。Linux 系统往往会立即从写请求返回,并将数据写到高速缓存中。因此,只要一个写请求的截止时间还没有结束,该基于截止时间的调度程序就会优先处理读请求。这是数据库系统优先选择的调度程序,特别是在高性能磁盘驱动器的情况下。

完全公平排队调度程序

完全公平排队调度程序 (complete Fair Queuing Scheduler, CFP Scheduler) 同样也集成了搭便车调度算法的请求合并处理机制及类似于向前看调度算法的单程电梯序列。除此之外,它试图赋予使用特定设备的所有进程以基于标准时间间隔的相同数量的同步输入/输出请求。对于多用户系统来说,它比其他调度程序的效率要更高一些,故而也是当前大多数 Linux 发行版系统中默认采用的磁盘调度程序。

14.7.9 向控制器发送命令

标记式排队 (tagged queuing) 最初是在小型计算机系统接口 (SCSI) 类型的磁盘驱动器领域中研制出来的一种技术,有时称之为命令式排队 (command queuing) 或本地命令式排队 (Native Command Queuing, NCQ),其基本的思想是把全部或部分的磁盘操作调度任务委托给磁盘控制器来完成。进一步说,此类驱动器的设备驱动程序把所有的输入/输出请求都直接地传给了磁盘控制器,而磁盘控制器则需要完成所有对输入/输出操作的调度。其理论依据是,磁盘控制器拥有磁盘几何结构相关的一个不同层级的信息以及磁盘机构的当前状态,故而可以更好地完成针对多个磁盘请求的调度任务。这种把功能向硬件迁移的现象,我们在操作系统领域中可以经常看到。一旦某种技术证实了在操作系统中很有用,我们就会开始考虑把相关功能植入硬件中,其间对应功能实现起来成本通常会更低廉,有时候也会更好一些,而且还可以把珍贵的处理器和内存资源解放出来。在这种情况下,磁盘控制器能够更好地完成相应工作,是因为它能够综合考虑和权衡旋转延迟的因素。当一开始利用这些磁盘

[321]

调度算法来完成大量调度工作的时候，寻道时间要远远地大于旋转延迟时间。然而，在过去的 20 年里，在寻道机构方面的改进已意味着现在的寻道时间与旋转延迟时间几乎没有什么差别了（如表 14-2 所示）。一般来说，操作系统并没有掌握多少关于磁盘驱动器的旋转位置的信息。除此之外，因为扇区分组机制和逻辑块地址处理环节，磁盘驱动程序甚至都无法理解磁盘驱动器真正的几何结构。然而，磁盘控制器则拥有所有相关信息，并因而能够利用一种综合考虑旋转延迟时间和寻道时间的算法来实现最优的调度。据报道，在许多情况下相关性已经提高了 30%，不过，这在很大程度上取决于特定情况和具体细节。在大多数的现代操作系统中，已经实现了标记式排队。同时，这项技术现在也被应用到了最新的高性能的高技术连接系统接口（ATA）类型的磁盘驱动器中。

14.7.10 哪种算法最好

在讨论了所有的这些磁盘调度算法之后，似乎理所当然地就想知道哪种算法才是最好的。遗憾的是，这个问题的答案依然是计算机行业中经常会听到的那句话——“要看情况而定”。事实上，没有一种算法在所有的情况下都是最好的。具体来说，先来先服务调度算法最为简单，而且消耗的资源最少。如果一个系统经常都是负载非常轻的情况，在队列中没有多少磁盘请求，那么所有算法的表现都相差无几——就像先来先服务调度算法一样。在这种情形下，其他算法没有一种可被证明是有道理的（译者注：言外之意，先来先服务调度算法当仁不让，应为首选）。

然而，许多系统经常都是中度甚至重度负载，因而我们不可能拿着如此简单的答案就能够抽身离开。在这样的场景中，先来先服务调度算法将会呈现出高度的服务可变性，反而往往是最糟糕的选择。于是，下一个需要提出的问题就是，我们试图要优化哪些参数呢？在大多数情况下，我们在尝试最优化磁盘的吞吐量。但是，我们之前就曾见到过，最大吞吐量的产生是以牺牲那些要访问不在最佳位置的文件的进程的公平性作为代价的。这些请求要么是遭受到响应时间的严重延迟，要么是遭遇到响应时间变化的不可预测，因此，都是无法被接受的。如果程序可以适时地提醒用户，那么用户还是能够容忍慢悠悠的响应的，但是，响应时间的巨幅变化会使程序变得不再可能足够有效地提醒用户。为此，在大多数情况下，向前看调度算法的一些变种算法则可能是最好的，同时这里假设你的系统里没有包含能够自己处理调度的新型设备。如果有此类硬件设备可以使用，那么它几乎肯定会比操作系统做得更好。

14.8 内存直接存取型控制器和磁盘硬件特征

因为磁盘控制器有一些特殊的硬件特征会影响到操作系统的设备处理程序的设计，所以我们就此在这里展开进一步针对性的讨论。

14.8.1 内存直接存取型控制器

最初，输入/输出控制器被设计成每次传递一字节（byte）或一个字（word）的数据。首先，处理器将会把控制信息加载到适当的寄存器中，这些控制信息包括操作类型（读、写或控制）、内存地址以及可能的设备地址。然后，处理器将会发出一条输入/输出指令。当输入/输出操作完成之后，控制器便会发出一个中断信号，于是，处理器就会开始策划传输下一个字或字节。这对于像键盘、调制解调器，甚至在非常早期的个人计算机上的早期的纯文本显示器（CRT）等设备来说，还是可以接受的，因为在下一个中断发生之前，处理器能够执行

非常多的指令（译者注：即处理器可以完成自己的本职工作）。然而，伴随设备变得越来越快，中断的次数和时限开始把处理器压得有点喘不过气来（译者注：即处理器就有点无暇顾及自己的本职工作了）。

322

为此，在计算机的输入/输出部件的设计中产生了一项新发明——内存直接存取（Direct Memory Access, DMA）型控制器。主处理器将向内存直接存取型控制器提供与之前加载到寄存器中的信息相同的信息，同时还会加上欲传输数据的长度（针对读或写操作类型而言）。内存直接存取型控制器将承担之前由处理器负责完成的工作，只是当设备控制器完成一个字的传输时，它将通告内存直接存取型控制器，而不是去中断处理器。伴随每个字节（或字）传输给内存或者从内存传输过来，内存直接存取型控制器将会对欲传输数据的计数值做递减操作，同时对相应的内存地址做递增操作。当欲传输数据的计数值变为0时，内存直接存取型控制器就会得知数据传输已经完成，这时候才会去中断处理器。这项技术极大地减少了处理器对输入/输出管理的开销。许多现代的控制器一般会在控制器内部拥有一个内置的内存直接存取型控制电路，而不是与其他的控制器共享一个内存直接存取型控制器。

14.8.2 磁盘驱动器的其他特征

现代的磁盘驱动器有一些其他的常见特征，会对操作系统产生影响。第一个特征就是磁盘驱动器中的缓冲机制。现在的小型磁盘驱动器的标准规格是包含8 MiB的内存（RAM），这部分内存可用作高速缓存（或称为高速缓存存储器）。在这种场景中，其也可称作磁道缓冲（track buffer）。当前，磁盘驱动器性能的主要限制因素是寻道时间和旋转延迟时间的组合。在我们对高速缓存的讨论中，我们总会提到空间局部性——其基本思想是指，当一个程序引用一块数据的时候，它极有可能很快就会去引用这块数据（即前面访问过的数据）附近的那些数据。如果我们正在顺序地读取一个文件，并且快速公平地处理相关数据块，那么我们可能会请求访问譬如5号扇区并开始对其进行处理。我们很快就会读完那个扇区，然后请求访问6号扇区。然而与此同时，6号扇区已经开始要通过磁头了，为此我们将不得不等待磁盘完完整整地旋转上一圈，之后我们才能够读取6号扇区。同样的遭遇也会在7号扇区上发生，以此类推。

因此，当我们向一部现代的磁盘驱动器发送一条命令，去读取磁道上的一个特定扇区的时候，一旦磁头位于相应磁道之上，对应磁盘驱动器往往会将整条磁道上的内容都读取到内存中。如果下一个出现的是10号扇区，磁盘驱动器将会开始读取10号扇区，并且会把整条磁道读完，直到环绕一圈读取到了9号扇区为止。然后，其将会返回我们所请求访问的那个扇区，同时把读入的其他扇区的内容存放到高速缓存中且尽可能长的时间，因为其认为我们可能很快就会请求访问其中的某些数据的。

14.8.3 扇区保留和扇区迁移

随着时间的推移，磁盘驱动器将会开始出现故障。不过，这个过程开始得很慢，最初可能是非常平缓的。偶尔，一个全新的磁盘驱动器也可能一开始就有几个坏的扇区。为了妥善应对和处理这些故障，磁盘驱动器通常在格式化时会保留一些“备用”的扇区散落在各个地方——也许每条磁道也就保留一个扇区吧。这些扇区起先阶段并不是编号方案的组成部分。相反，它们是以储备方式持有的，于是，当检测到故障的时候，系统才能够分配它们——也就是说，源自旧扇区的数据将会被复制到备用扇区中，同时备用扇区的编号亦需进行调整以

- [323] 匹配出现了故障的扇区。这时，故障扇区会获得一个不会被使用的编号。在某些情况下，驱动器硬件可以执行这项功能。而在另外一些情况下，则由设备驱动软件负责完成故障恢复和扇区迁移（或称为扇区重定位）。

14.8.4 自我监控报告技术

扇区保留是一种被动的技术，它解决了当我们发现故障时应当如何处理故障的问题。而自我监控报告技术（Self-Monitoring And Reporting Technology, S.M.A.R.T.）则是一种预测技术，其解决的是我们如何预见未来故障以及如何避免或缓解相关故障。自我监控报告技术是一种标准接口，硬盘驱动器可以通过该接口，向主机操作系统报告它的状态，并提供对未来故障日期的估计。借助于相关充分的预告信息，系统或用户就可以在驱动器故障发生之前对数据实施备份。自我监控报告技术是为高技术连接系统接口（ATA）和小型计算机系统接口（SCSI）两种平台定义的，由康柏公司（Compaq）开创，并正由磁盘驱动器制造商继续推动和发展。

鉴于驱动器架构会随着模型的不同而变化，所以自我监控报告技术针对磁盘驱动器的每种模型都包含了一组相应的参数。关于一种模型的故障检测的属性和阈值，对于另外一种模型来说，有可能是没有用的。为了拥有一个彻底全面的可靠性管理方案，磁盘驱动器必须能够监控许多要素。相关方案中，至关重要的因素之一就是要了解故障模式。一般来说，故障可以划分为两类：可预测的和不可预测的。

不可预测故障发生很快，像电子和机械类问题，譬如，电涌可能引发芯片或者电路故障。改进质量、设计、过程和制造能够减少不可预测故障的发生。例如，半钢子午线轮胎的发展就减少了旧式轮胎设计中常见的爆胎的次数。

可预测故障是以磁盘驱动器出现故障之前的某些属性伴随时间推移而呈现出来的退化为特征的。这样便创造了属性可以被监控的情景，从而使预测性故障分析成为可能。一般来说，许多机械性故障被认为是可预测的，例如，磁头飞行高度的降低将预示着潜在的磁头碰撞的发生。某些电子故障在故障发生之前也会出现属性退化的现象，但是更为普遍的是，机械问题是渐进式和可预测性的。例如，油位是许多汽车上可被监控的一个因变量或者说属性，当汽车诊断系统感觉到油位偏低的时候，油位指示灯就会变亮。于是，驾驶员就可以停下车来，以保护引擎。同样，自我监控报告技术可以为系统管理员提供充分的预告信息，让他们有足够的时间开启备份过程和保存系统数据。通常情况下可以预测的机械性故障，占到驱动器故障的 60%。这项数据表明可靠性预测技术具有很大的应用潜力。拥有了自我监控报告技术的系统，许多未来的故障就可以被预测到，从而也就可以避免数据的丢失。

14.8.5 展望未来

- [324] 如果处理器的性能按每秒每美元的操作数来进行度量，其在过去的 20 年里大体上以每年 100% 的速度递增。而在同一时间段里，存储一兆字节数据的成本从 70 美元降到了 1 美元。此外，磁盘驱动器的传输速率则从每秒传输 1MB 提高到了每秒传输超过 300MB。尽管如此，我们把磁盘作为辅助存储器的主要制约因素是平均寻道时间和旋转延迟时间。而在同样的时间区段里，二者分别仅仅降低了大约 10 倍。进一步说，旋转延迟时间受到了外层磁道的音速的限制，而且这种情况将不会改变。这就意味着，这两个因素现在可以完全主导用来随机访问硬盘上的任何特定信息所花费的时间。总之，相对于处理器来说，驱动器的速度

仍然是很慢的，这便加剧了计算机系统性能的失衡。

在操作系统方面，我们为磁盘驱动器投放了大量内存块用作高速缓存，以便使其速度看起来更接近于内存的速度。我们还开发了精巧的调度算法以优化磁头定位机制的性能。总的来说，我们耗费了大量的资源，来管理这些越来越和处理器不同步的设备。

磁带驱动器曾经是大型计算机系统中的标准的辅助存储设备。但是在20世纪70年代，它们被磁盘驱动器所取代，从而在这个角色上消失了。不过磁带并没有完全退出历史舞台，由于其低廉的存储介质成本，磁带驱动器于是降级成为第三级存储器。现在越来越明显的是，同样的事需要发生在磁盘驱动器上。尚不清楚什么样的新型设备将会取代磁盘驱动器，但是，我们在此做一个强有力的预测，在未来的5~10年，我们将会看到一类新的可以买得到的存储设备，同今天的磁盘驱动器相比，这种设备具有近似的随机延迟但更为低廉的成本。纯电子存储器和微机电系统（Micro Electro Mechanical System, MEMS）是两种可能的候选存储设备。结合了闪存和旋转式媒介技术的混合式硬盘（Hybrid Hard Drive, HHD）已经可以在市面上买得到了。Windows Vista 操作系统已经能够利用额外的闪存，以作为高速缓存的一个高速扩展部分。这一章中所介绍的大部分的技术即将面临被淘汰的命运，而我们必须得就如何使用辅助存储器进行反思。或许，我们将会做一些更像 Palm 操作系统所做的事情。

14.9 小结

在这一章中，我们介绍了低级输入/输出的管理，其中着重围绕辅助存储器和磁盘驱动器展开说明。接着，我们讨论了一些主要的输入/输出设备类型，以及它们彼此之间存在什么不同之处。我们描述了一些用于支持输入/输出设备的通用技术。然后，我们探讨了磁盘驱动器的物理结构，以及存储在磁盘驱动器上面的信息的逻辑组织。我们还讨论了廉价磁盘冗余阵列，即好多磁盘以特殊的构型组装在一起，用来实现更大的吞吐量及更高的可靠性。本章还阐述了非常重要的磁盘调度操作及其对性能优化的改良举措。我们还探讨了一种称为内存直接存取型控制器的特殊类型的设备控制器，其可以明显地降低处理器用于输入/输出操作的负载。此外，我们也讨论了磁盘驱动器的一些特征，这些特征往往会影响操作系统的行为、驱动器的可靠性等。

参考文献

- | | |
|---|--|
| <p>Hofri, M., "Disk Scheduling: FCFS vs. SSTF Revisited," <i>Communications of the ACM</i>, Vol. 23, No. 11, pp. 645-653.</p> <p>Iyer S., and P. Druschel, "Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O," <i>Symposium on Operating Systems Principle</i>, 2001, pp. 117-130.</p> <p>Love, R., <i>Linux Kernel Development</i>. Indianapolis, IN: Sams Publishing, 2004.</p> | <p>Patterson, D., G.A. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," <i>SIGMOD Conference</i>, 1988, pp. 109-116.</p> <p>Russinovich, M. E., and D. A. Solomon, <i>Microsoft Windows Internals</i>, 4th ed., Redmond WA: Microsoft Press, 2005.</p> <p>Teorey, T. J., and T. B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," <i>SIGOPS Operating Systems Review</i>, Vol. 6, No. 1/2, 1972, pp. 114-121.</p> |
|---|--|

325

网上资源

- | | |
|---|---|
| <p>http://www.osdata.com (操作系统技术比较)</p> <p>http://www.pcworld.com/article/18693/how_it_works_hard_drives.html (硬盘特征及体系结构)</p> <p>http://www.littletechshoppe.com/ns1625/winchest.html</p> | <p>(每兆字节磁盘驱动器价格)</p> <p>http://www.answers.com/topic/hard-disk (磁盘传输速率)</p> <p>http://en.wikipedia.org/wiki/RAID</p> |
|---|---|

习题

- 14.1 就应用在磁盘系统的双缓冲和高速缓存进行区别比较。
- 14.2 磁盘调度算法传统上忽略旋转延迟时间的理由在于，其与寻道时间相比是非常小的。这种说法是否正确？
- 14.3 简要给出柱面的定义。
- 14.4 下面的哪一种磁盘驱动器组织提高了性能，但是没有提供冗余机制？
- RAID 0
 - RAID 1
 - RAID 5
 - RAID 6
 - 上述所有的选项都要求有相同数量的磁盘驱动器
- 14.5 下面的哪一种磁盘驱动器组织提供了冗余机制，但是在额外磁盘驱动器方面付出了最高的代价？
- RAID 0
 - RAID 1
 - RAID 5
 - RAID 6
 - 上述所有的选项都要求有相同数量的磁盘驱动器。
- 14.6 RAID 6 与 RAID 5 相比，有什么优势？
- 它在读取多数据块时，要更加快捷
 - 它在写操作多数据块时，要更加快捷
 - 它可以承受同时损坏两个磁盘驱动器的故障情况
 - 它要求更少的额外的磁盘驱动器
 - 以上都不是 RAID 6 相对于 RAID 5 的优势
- 14.7 不管一开始选择的方向是向里还是向外，循环向前看磁盘调度算法寻道时所跨越的磁道数量都是大致相同的。这种说法是否正确？
- 14.8 在第 14 章的末尾，我们讨论了为提高磁盘系统能力而引入的几种机制。在这些机制中，一些是用于提高性能的，一些是用于改善可靠性的。下面哪一项不是用来提高性能的？
- 标记式排队（本地命令排队）
 - 磁盘（控制器）硬件缓冲
 - 动态内存访问
 - 扇区保留
 - 以上都是为了提高性能的
- 14.9 首字母缩略词 CHS 指的是什么？
- 14.10 个人计算机硬盘上的第一个扇区中包含什么？
- 14.11 如果说先进先出调度算法（FIFO）是最公平的（从定义上看是如此），那我们为什么不使用这种算法呢？
- 14.12 简要描述搭便车磁盘调度算法。
- 14.13 为什么要引入磁盘分区的概念呢？
- 14.14 在磁盘驱动器上，循环冗余校验码（CRC）或纵向冗余校验码（LRC）有什么作用？
- 14.15 在磁盘驱动器上，错误校验码（ECC）有什么作用？
- 14.16 循环向前看调度算法是如何在向前看调度算法的基础上改进的？
- 14.17 内存直接存取型控制器的主要优势是什么？
- 14.18 某些新的磁盘驱动器支持本地命令排队或者说标记式排队。这究竟是一种什么样的技术？为什么说它是一个进步？
- 14.19 某些新的磁盘驱动器支持所谓的自我监控报告技术（S.M.A.R.T.）。这又是一种什么样的技术呢？

网络、分布式系统及安全

我们在这一部分所阐述的话题并非在所有的操作系统教材中都会出现。有一位声名狼藉的计算机硬件系统官员曾指出，“网络就是计算机”。这是一种奇怪的论调，但是它确实凸显了我们现在对网络连接的重视，大多数计算机一般都会与其他计算机连接在一起，尤其是会连接到互联网上。为此，本书的这一部分将会围绕操作系统中涉及网络连接、分布式系统以及在有关情况下所出现的安全与保护问题等相关方面的内容展开讨论。

第15章主要介绍计算机网络连接的基础知识。这个话题本身就是许多计算机科学教科书的主题，也是一个非常活跃的研究领域，故而在这里仅做一些简明扼要的阐述。我们采用自顶向下的方法来论述大多数现如今所用到的硬件和协议。当然，互联网功能方面的内容会偏重一些。相关主题涵盖我们想要把计算机连接到一起以形成网络的理由、应用层协议、传输控制协议/网际协议、数据链路层、广域网、物理层和网络管理以及远程监控。

鉴于简单的单用户系统没有通过网络相互连接起来，故而往往不需要保护和机制，所以即使有，早期的操作系统也没有在这方面提供很多功能。然而，现在我们可以发现，许多机器都拥有多个用户，尤其是家里的计算机，大多数都会连接到局域网或者互联网上。因此，安全是当前一项非常重要的考量因素，而且我们将会在第16章就有关内容展开讨论，相关话题包括身份认证、授权和加密。

计算机联网后，我们很快就开始开发某些部分是运行在不同的计算机上的系统，即分布式系统。故而，这便成为第17章的主题。同样，这也是一个出现在许多书本和课程中的主题，并且目前许多研究工作正在这一领域热火朝天地开展着，所以相关论述也是非常简洁的，就像第16章的处理一样。有关内容主要包括通信、进程、命名机制、其他分布式系统模型、同步以及容错。

计算机网络

我们之所以学习操作系统，是因为如果对操作系统的有关功能和机理没有一个充分正确的理解和认识，那么就不可能写出大型的高性能的应用程序。公平地讲，今天我们学习计算机网络也是出于同样的理由——如今编写的大型应用程序，没有使用网络技术的情况非常罕见。故而，一般说来对操作系统是正确的，对网络而言也是正确的。进一步说，如果对基本的网络连接没有一个充分正确的理解和认识，那么就无法构建和部署大型的分布式应用系统。

本章首先针对我们想要把计算机通过网络连接起来的理由进行简要介绍和解释说明。接下来，在第 15.2 节，我们就传统上在计算机网络讨论过程中所采用的网络功能层次模型进行了阐明。进而，在第 15.3 节描述了应用层所采用的一些典型的协议。在第 15.4 节中，我们以传输控制协议 / 网际协议为例，具体讨论了传输层和网络层。在第 15.5 节，我们把以太网作为代表，论述了局域网中的数据链路层。其后，我们在第 15.6 节概要说明了广域网的数据链路技术，在第 15.7 节介绍了物理层所使用的技术，而在第 15.8 节则简要讨论了网络管理的相关主题。最后，我们在第 15.9 节对本章内容进行了归纳总结。

331

15.1 为什么要把计算机通过网络连接起来

我们之所以想要建立一个分布在网络上的应用程序，从更为详细的级别上而言可以归为几个原因。在计算机系统不断发展和日渐成熟的过程中，我们希望使用网络的最初起因是和共享访问昂贵资源联系在一起的。一开始，相应的资源也就是一台大型计算机（mainframe computer 或 mainframe），我们通过简易的终端而不是个人计算机对其进行访问，我们访问存放在大型计算机上的数据以及在那里运行的程序。之后，随着局域网（Local Area Network, LAN）的逐渐普及，我们开始使用局域网来访问其他的共享设备，譬如一个部门的文件服务器、一台价格不菲的激光打印机、一个调制解调器池以及附属的通信线路。这些资源如果要提供给每一个用户就会花费太多的钱，而且这些资源通常情况下也不会每时每刻都在使用。因此，让这些资源可以通过网络来进行访问，就把相关费用分散和均摊到许多用户身上。诚然，共享资源使用起来可能不如直接使用本地资源那样方便，然而大幅缩减的成本足以弥补这一弱点。另一个共享昂贵设备的具体例子是把各个系统备份到拥有附属三级存储设备如可能是磁带驱动器的机器上。

随着网络变得越来越普遍，某些可以利用网络来完成的特殊事情也逐渐明朗化。其中一项特殊的事情就是通过把几台较小型的机器以冗余配置的方式组合起来构建一个系统，那样，即便其中有一台机器发生了损坏，整个系统仍然能够继续工作，纵然是以一种降级退化的方式在运作着。

有些时候，我们会把一个进程的计算任务分配到多台计算机上以提高运算速度。我们把有关处理过程分解成能由每台机器单独处理的更小的模块。类似地，可以把多台较小型的机

器配置到一个系统中,而且伴随有关应用程序规模的扩展还可以加入其他的机器。例如,这便允许一家公司开始时只使用一台机器来运营网站,而伴随相关需求的增长,对应网站还可以成功添置新的系统。成本因素是采用系统聚集的方式提升速度的重要原因。在某些情况下,由于涉及大量的数据和处理工作,所以有些应用程序利用单个的大型系统根本就不可能完成。或者在某些情况下,尽管一台大型的计算机可以完成有关工作,但是因为成本问题而无法实现。然而,相关系统可以被设计成使用许多较小的处理器,从而化解这一成本难题。或许,“外星文明探寻”(SETI)项目可以算得上此类系统的第一个例子。该项目收集大量的射电望远镜的数据并将其通过互联网发送给有关用户,而这些用户自愿使用通常只在后台运行的“屏幕保护”应用程序来处理相关数据。这一应用程序旨在查找可能代表另一个星球上的智能生命体的信号。现在,“外星文明探寻”屏幕保护程序已经拥有数百万的注册用户。这被看作一个针对多个数据流实施并行操作的单一的、松散耦合的系统,故而成为整个世界上最快的超级计算机(supercomputer)。一个非营利组织是没有办法买得起拥有这样超然处理能力的系统的,因此,如果不采用这一技术,有关任务确实永远都无法完成。“外星文明探寻”是第一个这样的系统,不过今天已经有许多类似的处理数据的其他系统,正在从事密码学(cryptography)、脱氧核糖核酸(Deoxyribo Nucleic Acid, DNA)、数学、重力波(gravity wave)以及其他的科学项目研究。我们将在下一章讨论分布式系统时再次探讨这个话题。

[332]

经过近几年互联网的巨大发展之后,网络的深远影响已经变得非常清晰,即对信息访问的急剧增长。当前互联网相对快速的响应时间使以往并不经济的信息交换方式已变得切实可用,“远程办公”(telecommuting)即在家办公的设想就是一个例子。许多工作要求在员工之间开展频繁的、持续的互动。从某种程度上而言,这种互动可以通过电子邮件来近似实现,而更加紧密的互动则可以借助于即时通信软件——一种交互式“聊天”机制来实现。在另外一些情况下,这种交互可能要求语音通信(voice communication)、共享白板(shared whiteboard)甚至视频会议(videoconferencing)。现在,如果以较低的价格可以获得足够的带宽,那么所有这些都可以通过互联网来加以实现,而且可能让我们中的更多的人能够呆在家庭办公室而不是通勤到一个中心办公室来工作,至少在业余时间是这样。所以,网络能够减少通勤所必需的资源消耗(以及由此所产生的污染),从而有助于解决某些相关的社会问题。

另外还有一些通过互联网来共享信息的例子。我们有可能与住在世界上某个遥远地方的某些人展开合作来共同完成某个项目。例如,想想那些一起工作、创建Linux的各种实用例程库的人们,正是因为有了这些实用例程库,才使得Linux操作系统不再仅仅是一个有趣的内核实例,而是一个完整的系统。他们中的大多数人很有可能从来没有见过面,而只是通过互联网在一起工作罢了。

从一个不那么热门的层面上来说,想想互联网的普通用户。现在,我们中的大多数人每天都在使用电子邮件,并且经常利用网络资源来回答问题、查找人员信息、购买产品、下载软件更新代码、处理我们的个人银行和其他的金融交易,等等。如果没有互联网,此类事情将无法完成。尽管网络最初是由于某些其他的原因而存在的,但信息共享一直是其一项主要的功能。我们认为,在将来,有可能你运行的大多数应用程序将会以多个部分在多台主机上的方式来加以运行,故而,如果你对网络本身以及操作系统如何使用网络没有一些必要的理解,那么你将无法设计出完好的应用程序。

15.2 基础知识

15.2.1 相关模型

为了研究和实现网络，传统上，一些模型被建立起来，把有关主题细分成若干小的主题，并将它们部署到软件的各个层级上。在这些模型中，每一较低层级往往会为紧邻的较高层级提供某些服务。尽管通常情况下会就什么样的功能在什么层级执行制定出相当不错的协议，然而这些模型并非完美无瑕，故而也不会总是被严格遵循。由此导致的结果是，有时会发现某些功能出现在了多个层次上。例如，我们几乎在每一层级中都可以发现和找到安全功能。此外，在某些情况下，把一个低级层次上的网络协议拿来并使其作为另一更高层级协议上面的一层来加以运行可能非常有效和益处多多。在这些情况下，层次模型可能会变得相当混乱不清。这些模型在整理和组织我们的思路方面依然十分有用，而且大部分关于网络的文献都是围绕它们来进行组织的。因此我们将简明扼要地针对有关模型展开讨论。需要说明的是，操作系统软件通常就是按照这些层次方式来进行模块化设计的。

最广为人知的网络层次模型称为**开放系统互连模型**（Open System Interconnect model，OSI 模型），如图 15-1 所示。它是由国际标准化组织（International Standards Organization，ISO）建立的。该模型只是一种抽象设计方案，并没有考虑当时已有的任何协议，不过后来围绕这一模型曾经设计出了一系列协议。而在 GOSSIP 协议（译者注：一种去中心化、容错并保证最终一致性的协议）的大伞下，美国政府甚至还曾一度强制要求在由政府购买的所有计算机系统上实现这些协议。然而，有关努力并没有取得成功，故而最终被迫放弃。作为一种抽象模型，开放系统互连模型确实存在一些问题，最突出的问题在于，其中有两个层次，即会话层和表示层，几乎从来未曾被实现过。

根据开放系统互连模型，构建了用于描述业已存在的传输控制协议 / 网际协议簇（TCP/IP protocol suite）的另一种模型。这种模型主要集中在上面的几层（大多数是传输控制协议和网际协议），而几乎忽略了那些较低的层级。显然，这种模型认为硬件和驱动程序仅仅是商品而已，故而可以从供应商那里采购获得。

在本章中，我们采用了一种通用的混合模型，其大致由开放系统互连模型的底部两层和传输控制协议 / 网际协议模型的顶部三层组成，如图 15-2 所示。**物理层**（physical layer）定义了用于通信的实际媒介（medium，或称为介质）以及在相应媒介上获取信息的技术，有关媒介可能是金属电线或电缆，也可能是光纤或电磁信号。**数据链路层**（data link layer）负责访问共享的媒介，主要致力于把信息打包成独立的数据包并且仲裁对网络媒介的访问。现在，称为网桥（bridge）或交换机（switch）的数据链路层设备用来连接相应设备，并使每一对设备就像直接相互连接一样，所以介质访问仲裁功能在很大程度上已不再使用，而且这种趋势日渐加剧和愈发明显。**网络层**（network layer）负责为信息在互联网中流通选择相应的传输路径，这里的互联网是由许多通常基于不同物理层技术的网络所组成的复杂网络。**传输层**（transport layer）主要负责在两个网络实体之间创建一个可靠的连接，尽管并不是所有的应用程序都需要这样的一种连接或可靠性保证。最后，**应用层**（application layer）则由一台主机上的与（通常是）



图 15-1 开放系统互连网络层次模型



图 15-2 一种实用的网络层次模型

另一台不同主机上的进程进行数据交换的进程所组成。

对于连接到网络上的设备而言，每一层都会存在一个实体用来负责该设备在相应层级的功能。在物理层，相应实体位于硬件中。而数据链路层的一些功能可能也在硬件中实现。对于大多数设备而言，在其他层的实体都是软件。每个实体依赖于其下面一层的实体通过应用程序接口为其提供服务。相应地，每一实体将为其上面的一层提供服务。当发送方设备的一个数据包沿着各个层级从一个实体传递到下一个实体时，每个实体都会在该数据包的前面增加一小块信息，这块信息被称为报头（header）。例如，一个数据包（data packet）可能带有应用层报头、传输控制协议报头、网际协议报头、以太网报头以及物理层报头。而随着该数据包沿着接收方设备的协议栈向上传输，每个实体将会依次剥离掉相应层级的报头，然后将对应数据包继续传递给更高的层级。为此，这些报头承载了在发送方设备和接收方设备的对应层次的实体之间进行会话的功能。

334

把网络连接技术划分为若干层级既有好的一面，也有不好的一面。从好的方面来讲，小模块更容易被人理解，也更有利于开发和调试。如果开发出了改进的版本，相关模块也可以用新的对等的模块进行替换。不同的组织可以专门针对不同的层级开展研发工作，从而设计出更好的算法并给出更优的开发实现。另一方面，这样一来，我们清晰地规定各层级之间的接口以及发送方主机和接收方主机的实体之间的会话就显得格外重要。为此，产生了许多不同来源的标准。在某些情况下，我们拥有事实上的标准（de facto standard，或称为事实标准）：或者是某家厂商首先想出一种不错的方案，而后其他厂商纷纷效仿，或者是某些供应商和用户汇聚一堂，共同就某种标准达成一致。在另外一些情况下，我们则拥有法定的标准（de jure standard，或称为法定标准）：从技术上来讲，这些标准的背后往往带有法律的效力。进一步说，有关标准是由专业的、国家的或国际的组织，譬如电气电子工程师协会（Institute of Electrical and Electronic Engineers, IEEE）、美国国家标准化组织（American National Standards Institute, ANSI）以及国际标准化组织等制定的。在网络领域，许多标准具体是由互联网工程任务组（Internet Engineering Task Force, IETF）的成员制定的。每项这样的标准均被称为一项互联网标准草案（Request for Comment, RFC）。在 <http://www.ietf.org/rfc.html> 的网站上包含了相关文档。我们通常不时地通过 RFC 来定义互联网协议的某些方面。

15.2.2 局域网与广域网

可以采用若干不同的方式来看待各种可能的网络技术。而每种不同的观点均需阐明网络之间的差异以及它们的性能特征。我们要考虑的第一种主要特性是拓扑结构（topology）——什么是各机器之间的连接模式？理解网络的一部分困难来源于一个网络的物理拓扑结构可能跟它的逻辑拓扑结构并不相同。关于网络拓扑的第一种粗略的划分结果是局域网（Local Area Network, LAN）和广域网（Wide Area Network, WAN）[⊖]，[⊖]。一般来说，广域网中的

⊖ 一些关于网络的教科书还把城域网（Metropolitan Area Network, MAN）作为另外一种不同的类型，但是有关区别在这里并没有多大效用。

⊖ 一些机构或学术权威明确提出，局域网和广域网的区别是一个地理问题：局域网往往处于一个狭小的区域范围中，而广域网则分布在一个广阔的区域范围内。然而事实上，地理因素所导致的连接特征差异非常小。一方面，光纤分布式数据接口（Fibre Distributed Data Interface, FDDI）局域网能够覆盖的距离超过 100 千米。另一方面，在以往经常可以见到，两部调制解调器分别位于两台主机的顶层，并通过广域网机制以及一根一二英尺的电线相互连接，因为相关机制是两个系统唯一共同的接口。从技术上来说，这是一种广域网连接，但其显然不具备地理上分布的特征。

网络连接是点对点的。也就是说，当两个系统相连接的时候，通信只在这两台主机之间进行，而对其他任何主机是不可见的。因此，有关数据包中并不需要包含地址信息，因为只有一台设备来读取它们。而既然没有地址，所以就没有办法发送广播式数据包（broadcast packet，指一种面向连接到网络的所有设备的数据包）或组播式数据包（multicast packet，指一种面向针对某一特定传输流感兴趣的相关设备的数据包）。通常情况下，广域网连接采用全双工（full duplex）模式，这意味着两台主机可以同时进行传输。另外，由于连接可以同时从两个方向使用和开展工作，所以就不需要针对介质访问进行仲裁——一台准备就绪和即将传输的主机已经那样做了。

另一方面，传统上局域网采用广播连接方式。当两台主机在局域网中彼此通信时，它们的通信将会经由一些被诸多设备所共享的媒介。因此，在局域网中的设备通信就必须得和所有其他连接到有关媒介的设备共享访问对应媒介。既然相关数据包必须带有地址，故而就可以使用像广播或组播之类的特殊地址。最后一点，因为许多主机共享了单一的连接，所以对硬件来说就有必要对媒介实施访问控制。

交换（switching）是一种全新的技术，而且这种技术使局域网和广域网之间的区别变得十分模糊或不太明显。各台设备直接连接到了采用诸如以太网技术的网络交换机的端口上，而以太网技术原先被用来把相关设备连接到局域网上。然而，交换机读取数据包中的地址，并将有关数据包仅仅转发到与合适的设备相连接的端口上。为此，与交换机连接的设备并不会像先前的共享访问技术那样来共享媒介，所以相关连接可以采用全双工模式，而且只要在交换机的处理能力范围之内，任何设备都能够以网络全速进行信息传输，同时大多数设备都能够处理所有它们可以发送的信息。然而，数据包还是必须得包含地址信息，而且依然支持广播方式和组播方式。

15.2.3 拓扑结构

无论是广域网还是局域网，都存在各种各样的拓扑结构以支持诸多设备之间的连接。在广域网中，我们可以采用如下的任意拓扑结构实现主机间的成对连接：

- 线形网络（如图 15-3 所示）
- 树形网络（如图 15-4 所示，顶层结点是焦点）
- 星形网络（如图 15-5 所示，中心结点是焦点）
- 环形网络（如图 15-6 所示）
- 部分互连的网状网络（如图 15-7 所示）
- 完全互连的网状网络（如图 15-8 所示）

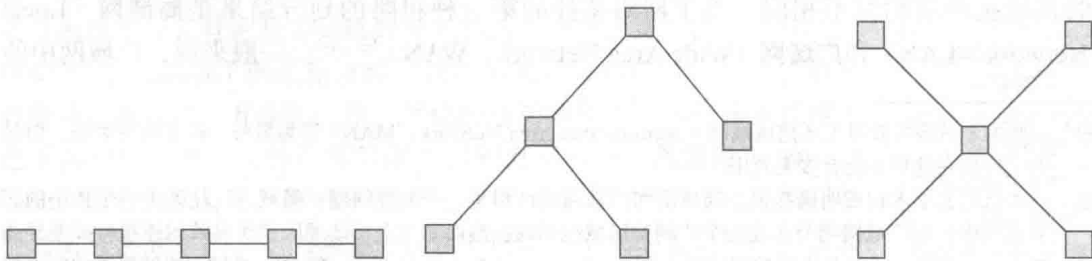


图 15-3 线形网络拓扑结构示例 图 15-4 树形网络拓扑结构示例 图 15-5 星形网络拓扑结构示例

上述每一种可能的拓扑结构均拥有一些与众不同的特点。首先，让我们考虑这些拓扑结

构用于广域网时的情形。相对而言,两台设备之间的每条广域网连接往往较为昂贵。进一步说,线形网络、树形网络和星形网络拓扑结构(star topology)的成本最为低廉,因为它们的连接数量最少。而就所有结点获取信息的路径而言,线形网络拓扑结构(linear topology)的信息获取路径最长,所以相应网络中到其他所有设备的通信可能会有点缓慢。当所有的网络设备都试图获取某种集中式服务时,往往会采用树形网络拓扑结构(hierarchical topology,或 tree topology)。这种结构在大型机时代非常典型。环形网络拓扑结构(ring topology)和网状网络拓扑结构(mesh topology)则比较可靠(假设有关通信可以在某个环上双向进行),因为其中通常存在冗余的路径可以进行通信。特别地,在环形网络拓扑结构中,唯一一条连接的失败并不会导致任何主机的通信失败。而完全互联的网状网络拓扑结构则是成本最高的,因为这种结构拥有非常多的连接。但同时,相关拓扑结构也是最快捷的,因为每一个结点和其他任何一个结点仅仅相隔一条连接的距离。而且,这种结构也是最为可靠的,因为失去一条连接只不过意味着失败连接两端的两个结点不得不利用一个中间结点来进行通信而已。部分互联的网状网络拓扑结构则是一种折中方案,故而相当常见,并成为互联网所实际采用的拓扑结构。对于存在冗余路径的网络来说,其数据链路层和网络层往往需要较为复杂的路由决策来实施数据包的处理。

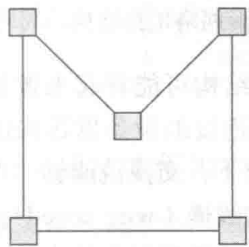


图 15-6 环形网络拓扑结构示例

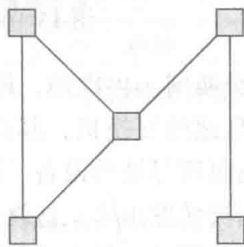


图 15-7 部分互联的网状网络拓扑结构

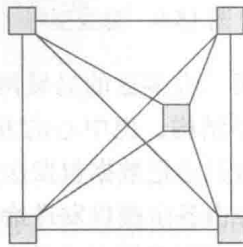


图 15-8 完全互联的网状网络拓扑结构

在局域网中,总线型网络与环形网络是最常见的两种拓扑形式。总线型网络拓扑结构(linear bus topology)看起来有点像图 15-3 所示的总线,但实际上是采用如图 15-9 所示的方式来进行连接的。在图 15-3 中,每一个结点与下一个结点之间都有一条连接,并且对于一端的一个结点和另一端的一个结点进行通信时,有关消息必须经由每一个中间结点来进行转发。在局域网中,采用的则是总线型网络拓扑结构,其中,总线是一种独立的介质,且每个结点都会与之相连接。为了使两端的结点能够进行通信,它们仅仅需要访问介质,然后就可以直接进行信息交换。从技术角度而言,采用环形网络拓扑结构的局域网的确是把消息在主机与主机之间实质性地进行了传递,只是大多数的主机并不会处理消息而已。对于这样的局域网而言,每个结点就像是连接到了非常类似于总线型网络拓扑结构的环上,当一台设备想要向另一台主机发送一条消息时,其仅仅需要等待轮到其自己而把消息传输到环上。接收方主机将会读取对应消息,而相应消息未曾寻址的主机则只需要把消息向前传送。

这两种技术存在一种可能的结合方式,进一步说,物理上而言是一种总线型网络拓扑结构,但是通过传送一个逻辑令牌(logical token)来控制对介质的访问,就好像对应局域网是环形网络拓扑结构一样。这称为令牌传送式总线,具体存在两种此类的协议实例。其一是附属资源计算机网络(Attached Resource Computer NETwork, ARCNET)协议,曾经广泛用于小型网络,但是如今主要限定在一些特殊的应用场合,譬如汽车内部。另一种是 802.4 协议,主要限于汽车制造产业,迄今未有进一步的发展。

337

在确定局域网的拓扑结构时，可能会产生一丁点的困惑。一个环形网络拓扑结构可能实际上会连接得像星形网络拓扑结构，如图 15-10 所示。位于图示中心的方框是一个中央连接结点，而有关媒介似乎是从每个结点连接到了中心的一个点。但是，该中心点并不是一个结点，并且信号实际上是以环形方式从一个结点传递到了另一个结点^①。类似地，总线型网络拓扑结构可以收缩成一个单一的集线器，就好像物理的星形网络拓扑结构或者树形网络拓扑结构，但是，有关信号是在同一时间广播到整个网络上的，所以相关电气连接采用总线型网络拓扑结构，而不是树形网络拓扑结构或者星形网络拓扑结构。从这种意义上而言，所有的结点都可以看到相应信号，但是并不需要将其转发到另一个结点。



图 15-9 总线型网络拓扑结构

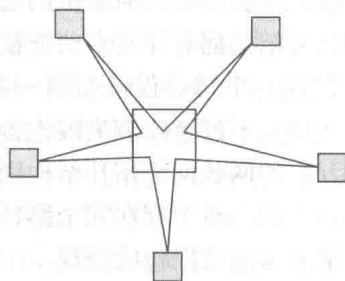


图 15-10 采用环形网络拓扑结构的局域网

今天，大多数的局域网其实都是交换网。再次地，网络拓扑结构可能看起来像是星形网络拓扑结构，但中心的方框是一个高速的交换机，其读取由所连接的设备发送来的数据包，并且只会把数据包发送给相应数据包所寻址的设备。通常情况下，交换机能够一次性地把所有经由各个端口发送给它们的数据流转发出去。这就是所谓的**线速**（wire speed）。如果交换机拥有许多端口或者这些端口是高速端口，那么这便要求相应交换机具有相当大的带宽（bandwidth）。我们将会在 15.5 节进一步讨论局域网与交换机。

15.3 应用层协议

15.3.1 应用层

在协议栈的每一层中，每台与网络连接的设备往往会包含相应的一个实体，且该实体与另一网络设备中的对应实体进行交互。在应用层中，在一个终端系统中便存在一个实体通过网络与一个服务器系统上的另一个终端应用程序进行交互。例如，有人可能在个人计算机上使用 Telnet 客户端与某一 UNIX 共享系统上的 Telnet 服务器进行通信。每个应用程序往往会使用一种特定的协议，且通常是相关应用程序专门设计的协议。有些时候，它们也会采用一种为服务于各种各样的定制式应用程序而设计的通用协议。在本节中，我们将简要地介绍几种应用层的协议，其中许多协议已经被广泛使用并且分配了**端口号**（port number）以供相应的服务器所使用。此端口号是提供给下一较低层级即传输层来使用的，用于确定哪个应用程序应当接收一条传入的消息。试想一部运行 FTP 服务器的系统可能还同时运行着诸如 Telnet、www 等其他服务。而源自网络的消息是随机到达的，所以每一层均需要在报头中包含发送实体应用于相应数据包的某些信息，以确定传入的数据包应提交给下一更高层级的哪一个实体。对于一个给定的应用程序而言，要使用的端口号可能是一个**标准端口**的

338

^① 中心的集线器可能包含一个结点以方便管理和数据收集，然而该结点并不是集线器功能的组成部分。

15.3.3 文件传输协议

文件传输协议 (File Transfer Protocol, FTP) 是另一种常见的应用层协议。与其他的协议不同, 文件传输协议使用两个端口而不是一个端口。20 号端口为主端口, 用于传输数据, 而 21 号端口也由文件传输协议所使用, 不过仅仅用来发送控制信息。这种设计允许一次较大的数据传输过程被中断, 例如, 在某个用户突然意识到当前正在下载的非常大的文件并不是所需要的文件的情况下。文件传输协议的另一个不同寻常的方面是, FTP 不仅是协议的名称, 同时还是使用这一协议的程序的名称。对应程序是一个命令行实用程序, 要想学会并熟练使用该程序并不是一件容易的事情。一种解决方案是使用保存好的脚本来运行该程序, 而另外一种常见的解决方案则是把该协议嵌入一个对用户更加友好的应用程序中。现在有许多图形化用户界面的实用程序包含了文件传输协议, 并可以支持文件的传输。当统一资源定位符是以 ftp:// 而不是 http:// 打头时, 大多数的浏览器甚至都能够支持和使用文件传输协议。

虽然文件传输协议所使用的命令是严格的 ASCII 码消息, 但是, 正在传输的文件则可能是程序, 并且经常包含有二进制数据。因此, 它们偶尔也可能包含看起来像是文件传输协议命令的字符串。这也正是数据传输使用了与命令通道相独立的一个通道的另一种原因。FTP 程序包括传输方式设置 (二进制方式或文本方式), 以便其可以传输包含任意二进制数据的程序和其他文件。

15.3.4 简单邮件传输协议、邮局协议及互联网邮件访问协议

简单邮件传输协议 (Simple Mail Transfer Protocol, SMTP) 用在电子邮件应用程序中, 以便从用户的电子邮件客户端程序发送电子邮件, 并将电子邮件从一个电子邮件服务器转发到另一个电子邮件服务器。很有意思的是, 电子邮件客户端使用了另外不同的协议来从服务器提取电子邮件, 具体通常为使用 110 号端口的第三版邮局协议 (Post Office Protocol version 3, POP3) 或使用 143 号端口的交互式邮件访问协议 (Interactive Mail Access Protocol, IMAP), (译者注: 现已改称为互联网邮件访问协议, 即 Internet Mail Access Protocol, 二者均简记作 IMAP)。邮局协议是一种较旧的协议, 虽然得到了广泛支持, 但是不太灵活。互联网邮件访问协议要更新一些且更为灵活, 但是尚未得到电子邮件服务器的广泛支持。

所有的邮件传输协议均使用纯粹的 ASCII 命令, 并且最初都是为了仅仅传输文本消息而设计的。ASCII 是 7 位代码, 而现代计算机通常都使用 8 位字节并忽略了附加位。一段时间以前, 相关需求却变得日渐清晰, 也就是说, 人们希望能够将所有各种类型的文件, 包括音频、视频文件以及二进制程序等附加到电子邮件中。所以, 对于这些附件来说, 附加位是不能被忽略的。为此, 针对简单邮件传输协议进行了扩充设计以方便处理其他的文件类型。具体而言, 多用途互联网邮件扩充协议 (Multipurpose Internet Mail Extension, MIME) 对简单邮件传输协议形成了补充, 并且支持在标准简单邮件传输协议的邮件内部封装非文本的信息。

出于传输可靠性的考量, 所有这些应用层协议在传输层上均采用传输控制协议。而其他不需要额外的传输控制协议可靠性的那些应用程序则往往采用用户数据报协议 (User Datagram Protocol, UDP)。特别地, 多媒体应用程序经常使用用户数据报协议, 因为它们不像传输数据的应用程序。也就是说, 流式多媒体应用程序通常不需要百分之百准确的数据传输。进一步说, 如果相关声音流未被高度压缩过, 那么声音流中丢失一个数据包往往是不会被注意到的。还有, 在应用层中, 这些程序大多使用专有的协议而不是互联网工程任务

组所审订的标准协议。

15.4 传输控制协议和网际协议

应用层协议由传输层上的协议所实现的相关实体来加以支持。也就是在几年以前，在传输层还有许多不同的网络协议组合（称为“套件”（suite））。然而，互联网的巨大成功却改变了这种情形。除了少数例外情况，所有的足够大到想要联网的计算机设施往往也希望连接到互联网。为了能够访问互联网，它们必须得使用传输控制协议和网际协议簇。当然，在它们自己的网络内部，它们也可以使用其他的协议簇。例如，如果想要使用互联网数据包交换（Internet Packet eXchange, IPX）协议来访问 Novell Netware 服务器，在计算机上除了需要加载传输控制协议，另外还需要加载互联网包交换协议，而这是一件非常简单的事情。当然，每个追加的协议簇同时也增加了复杂性，并且因为支持多个选项的成本较高，所以管理系统的人员常常希望尽量避免此类复杂性。为此，他们对系统供应商施加越来越大的压力以使其支持传输控制协议。故而，几乎所有的供应商现在都支持传输控制协议和网际协议簇。鉴于传输控制协议和网际协议簇所没有提供的重要服务，其他协议簇往往也没有提供，所以大多数网络管理人员已经弃用了其他协议。于是，传输控制协议和网际协议簇便开始主导网络连接领域。

15.4.1 传输层

在传输控制协议和网际协议簇中，网际协议是主要的网络层协议，而传输控制协议则是传输层上两种主要的协议中的一种，用户数据报协议（User Datagram Protocol, UDP）是另外一种。给定网际协议网络层的一个地址，网际协议将会尝试向对应地址传递一个数据包。用户数据报协议仅仅是把这项功能扩展到了应用层。这一非常有限的功能被称为“不可靠的数据报”（unreliable datagram）。在这里，“不可靠”一词并不意味着可能会失败，而只是说有关协议没有对传递做出任何保证而已。在许多情况下，这种“尽力而为”（best effort）功能特性其实就是所需要或所期望的一切。如果有关应用程序与连接的另一端上相应的应用程序正在交换消息，那么这两个应用程序往往能够判定是否发生了某种差错。例如，大多数的网络管理工具通常基于用户数据报协议来发送请求和响应。如果有关管理器在预定的时间内没有接收到特定的响应，那么它只需要重新尝试相应操作即可。

相比之下，传输控制协议则提供了“面向连接的可靠的”通信。给定一个网际协议地址和一个端口号之后，传输控制协议层将会尝试联系在对应地址的系统上的相应端口地址处所运行的实体，并与该实体建立一个连接。然后，它将会把数据发送给另一端的实体，同时接收响应和将之转发给主调应用程序，这一过程将会持续进行，直到应用层实体的一方中止该连接为止。这种协议采用了诸如消息编号和确认等各种各样的机制，以确保有关数据仅被传递一次，准确地来说是仅有一次被传送到另一端，并且是以这些数据被发送的顺序来进行传递的。另外，该协议还采用了其他机制来妥善处理由于网络拥塞等原因所造成的发送方传递速度太快（相对于接收方而言）的问题。

[341]

15.4.2 网际协议寻址和路由

如前所述，连接到另一台主机时，要求主叫主机应当知道目标主机的网际协议地址（IP address，或称为 IP 地址）。网际协议地址长度为 32 位。不过，当它们显示在人们面前时，

通常写成所谓点分十进制标记 (dotted decimal notation) 的特定样式。这种样式将 32 位划分成 4 字节, 并把每个字节显示为一个十进制数, 且彼此之间以句点相分隔。因此, 一个所有各位均为 1 的地址往往被写成 255.255.255.255。每个连接到互联网的网际协议网络均拥有一个与众不同的网络编号。在该网络编号内, 对应网络的管理员将为每台主机系统分配一个单独的地址。根据网际协议地址的哪些部分被用作网络编号以及哪些部分被用作主机地址, 网际协议地址曾经一度被划分为若干类别。这些类别具体被称为 A 类、B 类和 C 类地址。(也有用于特殊目的的 D 类地址和 E 类地址。) 1993 年, 这种机制被一种称为无类别域间路由 (Classless Inter Domain Routing, CIDR) 的新的机制所代替, 同时各类地址之间的技术区别大部分也已不复存在, 尽管人们经常还会将一个特定的地址归到这些类别中的某一类别上。

一类称为路由器 (router) 的设备负责把网际协议数据包从源设备传送到目标设备。每台路由器将会查看数据包中的网际协议地址, 并试图确定到达目标网络的最佳路径。因此, 路由器是根据网络层上的信息来做出把每个输入的数据包发送到什么地方的决定的。故而, 我们有时会说, 路由器是在第三层上做出转发的决定的。大部分的网际协议网络地址并不是按照地理位置进行分配的, 所以, 把网际协议网络连接到一起的路由器需要学会如何才能找到世界上的任何其他的网络。它们主要通过彼此之间的交流来获取有关的信息。它们使用各种各样的协议来进行信息的交换。两台路由器彼此间所使用的协议取决于它们之间的包括行政关系在内的各种关系。有好多这样的协议, 根据相应的基础算法可以被划分成若干分组。其中较大的分组是距离矢量算法群, 具体包括路由信息协议 (Routing Information Protocol, RIP)、第二版路由信息协议 (Routing Information Protocol Version 2, RIP2)、内部网关路由协议 (Interior Gateway Routing Protocol, IGRP)、增强型内部网关路由协议 (Enhanced Interior Gateway Routing Protocol, EIGRP) 以及边界网关协议 (Border Gateway Protocol, BGP)。而链路状态 (link state) 算法群中当前的一个主要代表是开放式最短路径优先协议 (Open Shortest Path First, OSPF)。

互联网中的路由器往往以拥有许多冗余链路的非完全网状拓扑结构相连接, 这样, 一条链路出现问题时通常就不会把整个网络分割成若干无法通信的片段。当然, 链路故障仍然可能导致服务质量的某种程度的下降或退化, 因为链路故障必然会使网络的某些部分不得不承担更多的负载。在早期的网络中, 术语网关 (gateway) 曾经被用来指代我们现在称作“路由器”的这一类设备。当在任何设备上配置网际协议时, 往往还可能见到这一术语的使用, 特别是作为短语默认网关 (default gateway) 的一部分。这里, 该名称本来是用来指定一台主机在不知道究竟应该采取什么样的最佳路径来实现其与另一台主机之间的通信的情况下所使用的本地路由器。确切地讲, 现在应当使用短语默认路由 (default router)。而术语“网关”, 准确地说, 则被用来指在应用层上运行的两个代理之间进行连接的服务。具体而言, 一个不错的例子, 如用来连接面向大型机的电子邮件系统 (譬如 IBM 的办公软件 OfficeVision) 以及运行简单邮件传输协议和第三版邮局协议的 TCP/IP 电子邮件系统的电子邮件网关。

毋庸置疑, 每台网络设备都可以按照预先确定的静态的网际协议地址实施特定的配置, 但是这样将会很难管理。当然, 凭借名称而在整个互联网上为人熟悉的各台服务器往往会拥有永久性的已分配的地址。然而对于其他的情况来说, 采取动态分配地址的方式则会容易许多。例如, 计算机经常会伴随人员在部门之间的调动而搬来搬去。而笔记本电脑则使得有关情况变得更为糟糕, 因为它们常常从办公室搬到了家里甚至进而带到了邻近的一个闹区。因此, 人们设计了一种协议, 即动态主机配置协议 (Dynamic Host Configuration Protocol,

DHCP) 来方便和适应这种移动性。于是, 每个网络管理员通常会搭建一台动态主机配置协议服务器, 并为该服务器配置上已分配给相应网络的一系列网际协议地址。一台刚刚打开的主机将会发出寻找动态主机配置协议服务器的广播消息。而动态主机配置协议服务器将会回复该主机, 并告诉它包括关于应当使用哪个网际协议地址在内的各种需要了解的信息。相应地址将会被对应工作站租用一段时间, 之后则必须重新进行申请。不过, 动态主机配置协议服务器也可以配置成每次都为特定的计算机提供相同的网际协议地址, 但是这通常仅适用于服务器、打印机以及其他的一般情况下不会经常发生变化的有关系统。

15.4.3 名称解析

鉴于人们发现要记住网际协议地址并不是一件容易的事情, 所以, 传输控制协议和网际协议簇便包含了把用户容易掌握使用的名称转换为网际协议地址的相关机制。用于实施这一转换的协议称为**域名服务** (Domain Name Service, DNS)。域名服务依赖于一系列按层次结构组织的服务器来完成有关转换。例如, 一台主机可能使用了域名服务器, 以便把本地网络上的名称 “webserv” 转换为一个网际协议地址。如果有关用户位于相应名称所处的域 (domain) 中, 那么域名服务器就可能会返回像 223.1.2.1 这样的网际协议地址。而如果有关用户位于相应名称所处的域之外, 那么要尝试找到这一相同的服务器, 就必须得使用该名称的另外不同的形式, 譬如 webserv.example.com, 称之为**完全限定式名称** (fully qualified name)。在这样的名称中, 句点之间的每个部分被称为域名。通常情况下, 有关的域名被组织成一个树形结构。各种各样的较高层级的域名由不同的机构所拥有和进行管理, 而**顶级域名** (Top Level Domain, TLD), 在前面的例子里即 “.com”, 则在互联网工程任务组的授权下进行管理。如果一台主机想要查找这样的完全限定名称, 它首先得向其默认的域名服务器发出请求。而相应服务器的网际协议地址要么是通过动态主机配置协议获悉的, 要么是在配置网际协议时通过手工方式配置到主机中的。如果对应的本地域名服务器并不知道 webserv.example.com 的网际协议地址, 那么它将会向域名服务层次结构中的下一级别的相应服务器发出请求。以此类推。最终, 对应的地址将会被找到并返回给最初发出请求的那台主机。

343

15.4.4 第6版网际协议

到20世纪90年代初期, 全世界的网际协议地址似乎很快就将用完了。于是, 这便极大地推动了对新版的传输控制协议和网际协议簇及网际协议地址的定义。相关新的版本称为第6版网际协议 (IP version 6, IPv6)。其中, 第6版网际协议将会支持非常多的网际协议地址, 进一步说, 在我们仍然使用传输控制协议和网际协议簇的情况下, 我们将几乎不可能用完相应的网际协议地址。与此同时, 若干事件的发生也减缓了网际协议地址相关爆炸性的增长需求。首先, 无类别域间路由能够支持对许多先前曾经分配给某些机构但那些机构不再需要的网际协议地址进行重新利用。其次, 在有关主机经常长时间地处于关机状态或者定期接入和离开网络的情况下, 动态主机配置协议便可以支持对网际协议地址的动态重用, 为此相关机构即便是拥有较少的网际协议地址也可正常应对网络连接需求。最后一点, 开发完成了**网络地址转换** (Network Address Translation, NAT) 技术。网络地址转换是一种新型技术, 其使用网络内的一组地址, 并把这些地址转换成为在互联网上本地网络之外的地方所看到的另外不同的 (甚至更少的) 一组地址。这些技术汇合起来, 便意味着升级到第6版网际协议的压力就大大消除了。尽管如此, 考虑到第6版网际协议的其他方面的功能, 这种迁移从长远来

看最终可能仍然将会发生。幸运的是，第 6 版网际协议被设计为支持飘逸得体的迁移方式。总的来说，大多数的路由器供应商已经支持第 6 版网际协议，并且大多数新版的操作系统也包括对第 6 版网际协议的支持，不过，似乎没有很多用户要迁移到第 6 版网际协议上。然而，有一种专门使用第 6 版网际协议的称为第二代互联网（Internet 2）的研究性网络正处于与互联网同步发展的过程中。

15.4.5 公共实用例程

许多实用例程通常和传输控制协议及网际协议栈一起结伴分布，其中有一些是设计用来访问公共服务的，例如：

- 访问超文本传输协议服务器（即万维网服务器）的浏览器。
- 访问文件传输协议服务器的 ftp 客户端（有时也由浏览器来完成）。
- 用作远程登录命令解释器的 telnet。
- 用于处理简单邮件传输协议、第三版邮局协议以及交互式邮件访问协议等电子邮件的 pine。

另外还有一些常见的分布式实用程序是为网络管理而设计开发的。有关工具的了解将有利于帮助任何系统设计人员理解本地网络的操作及其是如何影响系统设计的。这些工具将会在第 15.8 节中展开进一步的讨论。

15.4.6 其他协议

虽然在网络层运行的其他协议大部分都已经被放弃和退出历史舞台了，但是运行 IBM 的系统网络体系结构（System Network Architecture, SNA）及程序与程序间高级通信（Advanced Program-to-Program Communication, APPC）协议族中的协议的系统装机量仍然十分可观。其中一些协议在时间上要早于传输控制协议和网际协议栈。某些运行这些协议的设备是不可编程的，并且不能轻易升级。此外，这些协议具有一些特殊的功能，使得它们在高级别需求的情况下比较有用，因此它们可能在未来的一段时间内仍然将会继续使用。

[344] 另一种在过去十分常见的协议是互联网数据包交换协议（IPX），由诺勒公司（Novell）在其 Netware 服务器上使用而推广开来。互联网数据包交换协议拥有一项使其非常受欢迎的特征，进一步说，介质访问控制（Media Access Control, MAC）层地址被用作网络层地址的一部分，并且客户端系统可以自动获知对应地址的剩余部分。这意味着，即使相应系统迁移到了另外一个物理网络中，客户端工作站中的互联网数据包交换协议的驱动程序也无须对地址进行配置。这便大大简化了网络管理，这也可能是有关操作系统得以流行开来的一项重要因素。然而，互联网的普及最终盖过了这项优势，从而导致诺勒公司最终抛弃了该项协议。尽管如此，互联网数据包交换协议已经在线上多人网络游戏中找到了自己的合适位置，故而也有可能继续会伴随我们存在一段时间。人们还使用过各种各样的其他协议，大多数与特定的操作系统相关。有关实例包括在数字设备公司（Digital Equipment Corporation, DEC）的硬件上使用的数字设备公司分层网络体系结构协议（Digital Equipment Corporation Network, DECNet）和本地传输（Local Area Transport, LAT）协议以及在 Banyan 系统上使用的虚拟集成网络服务协议（Virtual Integrated Network Service, Vines）。网络基本输入/输出系统（NETwork Basic Input Output System, NetBIOS）协议最初是为 IBM 公司及其小型局域网而开发的，但该协议最后却被微软公司所采纳，并且只是在后来发行的高版本的 Windows NT

中才开始走向消亡。还有一些其他的供应商也已经被收购、合并或不复存在。在某些情况下,由 IBM 公司所开发并且由微软公司所广泛使用的网络基本输入/输出系统协议的某些残余或印迹依然存在。具体来说,其中就包括服务器消息块(Server Message Block, SMB)协议以及在 UNIX 和 Linux 操作系统中用来访问微软服务器的开源 Samba 软件包。尽管如此,最新发布的 Windows NT 系列操作系统已经明确表示,传输控制协议和网际协议簇将是它们的首选方向。

15.4.7 防火墙

遗憾的是,这个世界上总会存在一些缺乏相应知识、欠缺相关技能或者行为图谋不轨的人。于是,不好的东西便可能进入一个通过互联网(或任何类似的网络)而暴露在整个世界面前的网络中。鉴于此,人们已经设计和开发出了相关设备来保护网络免受有关恶意信息的影响。一般来说,这些设备就是路由器。这些路由器被放置在从互联网到相应网络的入口处,并像平常那样接受来自互联网的各种数据包。但是,在路由器将有关数据包转发到网络内部的局域网之前,它们将会执行一些额外的操作功能,仔细查验数据包内部,从而确认是否存在某些东西是相应的网络管理员已经设定不想让其通过路由器的。这些检查可以包括很多方面。这里仅举几个代表性的例子(我们将会在第 16 章展开具体的讨论):

- “拼接”测试(PING)
- 垃圾邮件(SPAM email)
- 病毒(Viruse)
- 已知的拒绝服务攻击(Denial of Service, DoS)
- 访问不受欢迎的网站(例如家长控制)
- 访问未使用的端口

15.5 数据链路层

起初,被称为局域网的网络类型具有一种特殊的性质:其中的数据以一种特别的方式进行传输,使得连接到相同链路的所有主机实际上均能够“看到”每一传输的数据流。通常情况下,每台主机被配置成为仅仅“读取”实际寻址到其自身的那些信息。经常应用于此类局域网的另一种说法是“多路访问网络”(multiaccess network)。由于许多这样的主机连接到一个物理介质上,所以必须要设计一种机制来支持有关主机对介质的共享访问。相关机制被称为介质访问控制(Media Access Control, MAC)。由这一名称衍生出来的另一个术语则是介质访问控制地址(MAC address)。每台主机都通过一个网络接口卡(Network Interface Card, NIC;有时也称为网络适配器,即 network adapter;简称网卡)连接到局域网上。每个网卡都拥有一个由相应制造商所分配的 6 字节地址,其中前 3 字节用来标识制造商,而后 3 字节用来标识对应特定的适配器。在大多数情况下,我们可以放心地认为这些地址在全世界是唯一的,尽管确有报道声称某些不守道德准则的不良供应商在假借其他供应商的标识编号进行网卡的制造。

对于介质访问控制功能来说,同样也存在许多不同的相互竞争的机制。但是,其中只有 4 种机制是非常成功的,这就是以太网(Ethernet)、附属资源计算机网络(Attached Resource Computer NETwork, ARCNET)、令牌环(token ring)和光纤分布式数据接口(Fiber Distributed Data Interface, FDDI)。附属资源计算机网络是第一代局域网技术之一,但是其

具有非常大的局限性。然而，正是有关限制约束转变成为其在嵌入式系统中的优势，于是附属资源计算机网络技术当前仍然还在此类场景中继续发挥着作用，但是从其作为一种一般的局域网技术来说，它却实际上已经消失了。以太网和令牌环最终由电子电气工程师协会（Institute of Electrical and Electronic Engineer, IEEE）分别审定为 802.3 标准和 802.5 标准。而光纤分布式数据接口则成为美国国家标准化委员会（American National Standard Institute, ANSI）确立的标准。尽管我们常常会简单地说起“以太网”，但其实这并不非常准确或正确，之所以这样说只是因为这种叫法已经成为一种相当普遍的惯例而已。以太网具有一项突出的优点，那就是无论同令牌环还是光纤分布式数据接口相比，它都是一项更为简单的技术，故而通常更容易实现正确安装并且也更为便宜。当然，以太网也存在一系列的严重的缺陷或不足。

15.5.1 以太网

以太网建立在大多数时间有关网络并不繁忙的前提条件之下。如果网络比较繁忙，那么发送方就会等待，一直要等到网络变得空闲，然后才会发送。如果两个站点同时开始传送信息，那么它们的传送将会相互干扰，并导致冲突（collision）。以太网的介质访问控制机制被称为载波侦听多路访问 / 冲突检测（Carrier Sense Multiple Access/Collision Detection, CSMA/CD）。这一机制会引发两个主要问题。其一，有关带宽并非完全可用的。在负载繁重的网络中，吞吐量（throughput）在大多数情况下往往最大只能达到 40%~50% 的利用率。其二，如果有关网络超过这一限值，那么它最终将会到达一个冲突时时发生而网络完全停止数据传输的状态。而令牌环和光纤分布式数据接口则不会遭受这些问题。它们是确定性的，而不像以太网那样是随机性的。当一台设备添加到局域网时，单台主机的平均响应时间将会以一个可预测的量级发生下降。同时，每个站点往往会获得均等的访问机会，因此，很容易就可以让局域网运行达到非常接近于 100% 的利用率。诸如银行、医院和警察局等不能容忍故障并且需要能够预测响应时间的机构，通常会不惜花费多余的钱来使用令牌环或光纤分布式数据接口。

15.5.2 桥接与交换

为支持以太网克服上面提到的有关问题，人们最终开发出了一种解决方案，具体就是采用交换机（switch）替换掉了共享式接线集中器（shared wiring concentrator，又称为集线器，即 hub）。集线器是一种简单的物理层设备，它只是把源自任何一个端口的输入信号重复输出到所有其他的端口上。而交换机则是一种多端口设备，其只会把一个数据包发送到通向对应指定接收设备的那个端口上。在这一层级上所使用的地址是网卡的以太网地址。因此，这种转发决策是在数据链路层（或介质访问控制层）上来完成的，故而有时被称为第二层交换（layer two switching）。进一步说，此类交换能够在所有端口上同时接受和转发输入（不过会受到相应的交换机背板的限制）。此外，新方案还支持对应连接设备以全双工模式（full duplex）运行，也就是同时发送和接收，并以 10Mbps（兆位 / 秒）或 100Mbps 的速度运行。事实上，新型装置甚至可以自动感应相关交换机及所连接设备的最佳运作模式，从而使有关安装就像集线器那样简单。现在，廉价的交换机通常也能够让相应端口运行在千兆以太网（Gigabit Ethernet）的速度级别上。相关一系列的技术革新，使得以太网从最高吞吐量 5Mbps 的系统，发展到了相当普通的一台交换机便能够以 100Mbps 的速度提供 1Gbps（千兆位 / 秒）

的吞吐量的系统。对于更大更昂贵的交换机，则甚至能够提供更高的性能。

在多端口交换机变得司空见惯之前，当时称为**网桥**（bridge）的小型交换机，曾经被用来把大型的局域网划分成若干小的网段。把网络划分为若干更小的网段将可以允许每个网段上的设备支持更好的吞吐量（throughput）和响应时间（response time）。所谓网桥，就是最初只有两个端口的设备。它们通过读取与其相连接的局域网上的所有信息流，从而获悉通过每个端口可以到达的介质访问控制地址。当它们看到一个端口的某数据包发往指定地址的某台设备并且它们已经获知通过另一个端口能够到达相应设备的情况下，它们将把该数据包从对应另一个端口转发出去。这被称为**透明网桥**（transparent bridge）或**学习型网桥**（learning bridge）。如果两个网桥在两个局域网之间采用并行的方式相连接（这往往是期望中要做的事情，因为这样将会提供冗余链路以防备出现某个网桥发生故障的情况），那么透明网桥便可能会出现一个问题。也就是说，有关网桥将可能形成一个环路，并且有关数据包沿着相应环路持续不断地进行传输。鉴于此，人们针对透明网桥开发出了一项功能，即允许它们以并行方式进行连接（甚至可以连接形成更为复杂的网状网络），但却不再构成环路。有关网桥彼此之间相互协调，并且通过不在所选择的路径上转发数据流，从而形成一棵**生成树**（spanning tree）。这棵生成树将会把数据转发到任何地方然而却不会包含任何环路。如果一个网桥（或一个端口）出现了故障，那么相关网桥将会察觉到这种情况并且会形成新的生成树。这种解决方案的最大问题将会出现在某些连接并非局域网连接而是广域网连接的情况下。因为，广域网的连接线路相当昂贵（相对于其他网络的成本而言），并且拥有一个把广域网的路径切断以防止形成环路的网桥是非常奢侈的，一般很少有人能够承担得起。

当多端口交换机刚刚上市的时候，它们通常就可以把来自所有输入端口的信息流快速地转发到输出端口，就像由所连接的主机发送的速度一样迅捷。这被称为**线速**（wire speed）转发。有关营销人员希望把这种行为同早期的网桥区分开来，所以他们采用了**交换机**（switch）这个词。鉴于其出色的性能，多端口交换机开始切入和瓜分路由器的市场。通过采用**专用型集成电路**（Application-Specific Integrated Circuit, ASIC）和重新设计路由器，工程师便构造出了能够在网络层完成转发决策但却可以像第二层交换机那样以线速快捷运作的设备。于是，有关营销人员再次介入，他们把这些新型的高速路由器称为**第三层交换机**（layer three switch）。

347

15.5.3 令牌环

令牌环硬件具有一套与以太网完全不同的介质访问控制机制。有关硬件采用一种称为**令牌**（token）的特殊的空的数据包，该令牌持续不断地在主机之间进行传递，直到它到达一台需要传输数据帧的主机。此时，对应主机将会把该令牌更改为一个数据帧，进而把它发送出去。虽然这似乎有些效率低下，但实际上效果却很好。如前所述，令牌环硬件可以轻松达到98%的带宽利用率。相比之下，共享式以太网很少能够达到60%的带宽利用率，并且通常甚至要低不少。

令牌环网桥可以采用与以太网网桥相同的方式运作，但是它们还支持一种被称为**源路由桥接**（source route bridging）的更复杂的模式。在这种模式下，连接网络的有关主机将会通过网桥获知路径，并且每个数据包都将包含此路由信息。这种方案具有如下几个优点：

- 网桥无须了解地址，并且更简单、更低廉。
- 网桥可以按网状方式进行连接，故而可以利用所有链路。

- 跨冗余链路的负载往往会实现自我平衡。

但遗憾的是，源路由相关功能要求得实施某些配置（相对于透明网桥来说，后者基本上不需要任何配置）。同时，它们是利用广播方式来寻找首选路由的，故而经常会被指控造成了广播风暴（broadcast storm）。当以太网通过利用快速交换技术克服了自身问题之后，令牌环选项连同源路由桥接技术便一同败下阵来。

15.5.4 其他的数据链路方法

光纤分布式数据接口是一种最初开发来用于光纤而不是铜线的技术。所以，从本质上讲，相关构建和安装的成本比较昂贵。当时，光纤分布式数据接口的运行速度便已达到了 100 Mbps，这要远远早于快速以太网能够达到这一速度之前。同时，光纤分布式数据接口环的周长可以超过 200 千米。不过，考虑到成本因素，光纤分布式数据接口通常不用于各台主机的连接，而只用于主干（backbone）局域网实现一个大院中建筑物之间的网桥、交换机或路由器的连接。后来经过修改，光纤分布式数据接口也可以运行在较短距离的铜线上。

还有许多其他的技术也参与了局域网领域的竞争。其中获得有限成功的一项技术是异步传输模式（Asynchronous Transfer Mode, ATM）。就像令牌环一样，异步传输模式是一种相当复杂的技术。然而，异步传输模式提供的功能使得其在某些场合颇具吸引力，比如说希望在单一网络上把数据传输及语音和视频传输混合进行并且要向所有用户保证最合适的服务质量（Quality of Service, QoS）的情况下。显然，异步传输模式在广域网领域一直是赢家。在局域网领域中，提供不同服务质量要求的服务的目标是通过网络的过度建设来实现的，故而任何应用都可以拥有其所想要的任何服务。因为当前的带宽成本非常低，所以这是可能的。如果使用异步传输模式来进行端到端的通信，将会实现最佳的总体性能。在利用异步传输模式的地方，其成功率还是很高的。在这一点上，异步传输模式是否会成为局域网领域的一个主要因素尚不确定。

附属资源计算机网络也曾经非常流行。它是美国国家标准化委员会的一项标准，但不是电子电气工程师协会的标准，并且已退出历史舞台。其失势的主要原因在于，它不支持桥接，同时，它拥有一个网络层地址，该地址只有一个字节，由硬件交换机进行配置而且相关配置过程非常容易出错。

348

15.5.5 网际协议地址到介质访问控制地址的映射

我们之前曾经提到，人们往往引用的是像 `webserv.example.com` 之类的便于人们记忆和使用的名称，并且网络层采用了域名服务来把对应的名称转换为一个网际协议地址。我们还曾说过，在局域网上，信息实际上是通过指定介质访问控制地址来进行发送的。那么，一个显而易见的问题就是，有关软件是如何把网际协议地址映射到相应的介质访问控制地址的？而答案是，这里使用了一种称为地址解析协议（Address Resolution Protocol, ARP）的特殊协议。正在查找服务器的一台主机往往会生成一个包含主机和服务器的网际协议地址的地址解析协议数据包。同时，在其介质访问控制报头中，将会包括对应主机自身的介质访问控制地址。但是，因为有关主机尚不知道相应服务器的介质访问控制地址，所以该主机将会利用一个广播式介质访问控制地址（即所有各位均为 1 的地址）把对应数据包发送给所有主机。每台主机将会读取该数据包，然后把它传递给网际协议软件。而网际协议模块将会把该数据包进一步传递给地址解析协议模块。指定的网际协议地址所对应的服务器中的地址解析协议

模块将会识别出有关主机正在通过地址解析协议进行寻址定位,从而会准备一个响应地址解析协议的数据包。这个数据包将被直接发送回对应正在查询的主机,于是,接下来该主机就可以采用相应的介质访问控制地址继续以后的会话过程。这时,除了被指定的主机,所有其他主机中的网际协议软件将会忽略相应的数据包。此外,在一般情况下,有关操作系统通常会将相关的介质访问控制地址缓存到一张地址解析协议表(arp table)中。

15.5.6 面向硬件的功能迁移

伴随网络的日渐成熟,某些最初由设备驱动程序在软件中所完成的功能已经被迁移到了相应的硬件中。这一演化步骤往往会需要一些时日,因为一项功能在被完全搞清楚之前,是不应当被迁移到硬件中的,毕竟硬件错误的修复成本是相当昂贵的。进一步来说,在网卡中,有两个对应功能被迁移到硬件的例子。其一是循环冗余校验(Cyclic Redundancy Check, CRC,或称为纵向冗余校验码)的计算。循环冗余校验是针对通过网络传输的数据块进行计算的一类功能,具体在第 14 章中展开了较为详尽的讨论。循环冗余校验结果往往与对应数据块一起进行传输,并且接收方将会完成与发送方相同的计算。如果接收方所计算出来的循环冗余校验结果与伴随相应数据包一起发送过来的循环冗余校验结果不相匹配,那么接收方就可以获悉其间发生了一个错误。原先这项功能是由有关网卡的软件驱动程序来计算完成的。从处理器周期利用角度来说,有关计算成本还是相当昂贵的。然而,硬件工程师发现了一种相当普通的在数据包传输时来完成同样计算的途径。这是一种非常低廉的方法,可以为处理器减轻相当大的负载。尽管这项功能对于现代机器而言可能不会产生多大的区别和效果,但在该项功能开发时,有关机器运行速度非常缓慢,故而不失为一种合算的解决方案。

多播地址(multicast address)的识别是迁移到网卡硬件的另一项功能。一般来说,多播数据包在整个网络中进行传输,并且每块网卡都会看到它们。在任何一个时间,在给定的局域网上都可能有多条多播数据流在被使用。而特定的系统可能会对特定的数据流感兴趣,也可能不感兴趣。关于多播的一个很好的例子是股票报价应用程序,该程序通常可能在股票经纪业中运行。尽管并非所有的系统都需要查看相应的股票行情数据流,然而许多经纪人可能想要查看股票报价,因此他们将会运行一个特定的应用程序,以查看那些分配给对应数据流的特定的多播地址。曾经有一段时间,所有的多播数据流都会被所有的网络适配器所接收到,并将其向上传递给网络层。在那里,如果系统判定对相应的数据流不感兴趣,那么它们可以被丢弃掉。显然,这对于那些针对任何数据流都不感兴趣的系统来说尤其是徒劳的,因为它们也会被每个多播数据包所中断,并且有关软件还必须得检查数据包中的地址。鉴于此,这项功能最后也被迁移到了网卡上。有关协议栈(protocol stack)将会把任何使应用程序感兴趣的多播地址通知给对应的网卡。于是,发给这些地址的数据包将会被传递到协议栈,而任何其他的多播数据包将会直接被对应的网卡丢弃掉,这样对应的处理器也不会再被中断了。

[349]

15.6 广域网

与局域网中的主机通常放置在同一栋建筑物或至少是在同一大院中的情况不同,广域网是有关数据必须经由串行的点对点的连接进行传输的设备之间的连接。这些连接常常是在两部网桥或路由器之间,但有时也会是一台主机连接到一部网桥或路由器上,特别是当对应连接是拨号链路(或称为拨号连接)的情况下。对于经由普通老式电话服务(Plain Old

Telephone Service, POTS) 的拨号链路而言, 最高可用速率为 56 Kbps (千位/秒)。当一条广域网链路是永久性的连接的情况下, 该链路被称为专用线路 (又称为租用线路)。这些线路往往是数字信号的 (而不像拨号链路, 后者是模拟信号的)。通常情况下, 最慢的专用线路的速度是 56Kbps, 但是 64Kbps 也很常见。T1 线路是现有的另一速度的线路, 以 1.544 Mbps 的速率运行。有时, 也会有一些线路提供的速度在 64Kbps 和 T1 之间, 它们被称为次 T1 线路 (fractional T1 lines, Frac-T1)。另外, 还有一些更高速度的线路, 它们的速度是 T1 速度的数倍。T1 线路最初是为承载语音业务而设计的, 有关呼叫是已被数字化为 64Kbps 信息流的模拟信号。通过同步时分多路复用 (Time Division Multiplexing, TDM) 技术, 多达 24 条这样的慢速数字信息流可以组合到一条在电话公司交换中心之间运营的 T1 线路上。

15.6.1 帧中继

当使用广域网线路构建大型网络时, 线路总成本的一个重要因素是从客户驻地到当地电话公司办公室的电路部分——称为最后一英里 (last mile)。如果相应客户拥有多条租用的 56Kbps 的线路从家庭办公室连接到了不同的站点, 那么他们通常可以把这些线路按 24 条一组的方式多路复用到单条 T1 线路上。鉴于 T1 线路往往可以运行在一条标准的双绞线铜线上, 所以这样将会节省大量的线路成本。人们甚至还开发出来了另外一种技术, 在这个方向上更前进了一步。具体而言, 有关线路没有采用上面所描述的同步时分多路复用技术, 即不是发送数据流而是发送数据包, 同时每个数据包在通过网络时单独寻址和进行交换。这种技术被称为帧中继 (frame relay)。因为一些单独的 56Kbps 的电路的容量常常没有得到充分的利用, 所以这样做是非常有意义的。利用同步的时分多路复用电路的网络一般被设计成是针对接近负载业务峰值速率时的情况。然而, 由于最坏的情况往往不会经常出现, 所以通常总会存在未被使用的带宽。为此, 当采用帧中继技术而不是时分多路复用技术时, 一条 T1 线路实际上可以承载 40~50 条以 56Kbps 的速率运行的线路上的所有的数据包。或者, 单一的到运营商办公室的 56Kbps 的帧中继电路可以承载 3~5 条并非总是大量使用的 56Kbps 的电路上的所有帧。无论哪一种方案, 帧中继网络都可以为他们的用户节省大量的钱财。

15.6.2 其他广域网技术

一种曾经风靡一时的局域网协议是综合业务数字网络 (Integrated Services for Digital Network, ISDN) 协议。一根铜线电路可以引到家庭或小型办公室, 且可以承载两个 64Kbps 的信道^①。这种类型的服务被称为基本速率接口 (Basic Rate Interface, BRI)。这些信道可以各自承载单个数字化语音呼叫或一个数据信道。而两个数据信道也可以从逻辑上组合到一起从而用作一个 128Kbps 的信道。这实质上要比普通老式电话服务的线路更好。关于综合业务数字网络的一个巨大吸引力来自于网络的核心, 对应接口是一个主速率接口 (Primary Rate Interface, PRI), 其可以承载 23 个 64Kbps 的信道再加上一个用于信令的 64Kbps 的信道。主速率接口的主要优点在于, 有关呼叫可以是源自综合业务数字网络的基本速率接口设备的数字信号呼叫, 也可以是源自常规的调制解调器的模拟信号呼叫。其间, 模拟信号呼叫将由相应运营商在其办公室进行数字化处理后并以数字方式进行传送。今天, 在电话支持办公室中仍然使用综合业务数字网络的主速率接口服务提供纯语音的传输业务。

正如前面所提到的, 异步传输模式是另一种专门为广域网而开发的技术。异步传输模

^① 从技术上讲, 还有一种 16Kbps 的信道, 主要用于网络信令或诸如信用卡授权等低速类应用程序。

式就是被运用到极致的帧中继技术。进一步说,异步传输模式的本质在于,所有的信息流都被划分成小碎片,即48字节的信元(cell)。这些信元可以快速地、低成本地进行切换,并且可以准确地为每个用户提供其所需要的传输业务服务类型。而这正是相关运营商非常期望的,因为他们拥有一系列提供从电传业务到超高速数据电路等各种不同服务类型的合同。利用异步传输模式,他们实际上只需要部署一套网络,而只是在网络的入口和出口位置使用不同的设备,并使其看起来就像用户签约的服务那样即可。有关运营商只需要培训操作人员和技术人员维护一套网络,他们仅仅需要一类管理软件,以此类推。为此,不难看出为什么许多广域网的主干网络已经采用了异步传输模式。

对于家庭和小型企业而言,还有两种参与竞争的用于高速广域网服务的其他技术,即有线电视网络调制解调器(cable modem,或称为有线调制解调器或电缆调制解调器)和数字用户线路(Digital Subscriber Line, DSL)。这两种服务在提供一些其他服务的同时,并利用异步传输模式及类似技术来提供到互联网的永久性连接。就有线调制解调器情形而言,对应的其他服务最初是有线电视服务。对于数字用户线路情形而言,对应的其他服务则是普通老式电话服务。鉴于采用了异步传输模式技术,有线调制解调器也可以用于传送普通老式电话服务,不过这是人们后来对原有概念的补充。另外,随着光缆(fiber optic cable,或称为光纤电缆)进一步延伸到各当地社区,每个客户的可用带宽正在不断加大,并且光缆最终应该会直接到达各家各户或每个办公室。这一技术的发展伴随有许多不同的名称,且大体上类似于“光纤入户”(Fiber to The Curb, FTTC)这样的提法。

351

15.7 物理层

为了让信息从一台设备流向另一台设备,必须要通过某种介质把两台设备连接起来。有关介质必须能够以某种方式发生改变,且相应变化可以被另一台设备所感知到。在历史上,对于计算机数据来说,这便意味着是某种类型的金属线把两台设备连接到一起并在其间传导电流。而在最近的这几年里,铜线常常被传导光的玻璃或塑料所替代。与此同时,通过电磁传输的无线传输也经常被用来发送信息。几十年来,无线传输曾经只是用于模拟音频和视频信号的传输以及文本电报的传输。但在最近的这几年中,无线传输已经更加普遍地被用来进行数据传输。最初,其被用于模拟数据的数字化传输,然而现在,正广泛地用在数据传输领域,这对于笔记本电脑和手持式计算机来说特别常见。

15.7.1 铜线规格

通信布线中所采用的金属线通常是铜线,并且每条线路往往由两根导线组成。有时,这两根导线是相同的,并且相互缠绕在一起,故而被称为**双绞线**(twisted pair)。一般而言,绞合在一起的导线既不太可能从其他导线摄取辐射信号,也不太可能辐射出让外部可以捕获的信号。同时,在两根导线(或组合到一起作为一条电缆的若干对导线)的外面还会缠绕上一层箔,这被称为**屏蔽型双绞线**(Shielded Twisted Pair, STP)布线方式。如果没有这种保护,则被称为**非屏蔽型双绞线**(Unshielded Twisted Pair, UTP)。与非屏蔽型双绞线相比,屏蔽型双绞线更不容易受到外部的干扰,也更不容易被网络外部截获信号。历史上在家庭和企业中安装电话线时所采用的电线就是某种类型的双绞线。用于数据传输的非屏蔽型双绞线电缆要求拥有比标准电话线更高的质量。非屏蔽型双绞线布线的质量一般按照美国**通信工业协会**(Telecommunication Industry Association, TIA)的类别(Category,简记作Cat)进行规范和

标准化。目前批准用于新建数据设施的最低类别是 Cat 5，对应速率大约为 100Mbps。而最新的标准则是 Cat 6，对应速率为 250Mbps。下一步将会推出 Cat 7 标准，即在 100 米的铜线电缆上运行 10Gb（千兆位）的以太网。

另外，还有一种称为同轴电缆（coaxial cable, coax）的铜制线路配置。对于这种情况而言，是在单一的中心导体外面包裹上绝缘材料。然后，在相应绝缘层的外围再编织上一层非常薄的线路。（偶尔，外面的这一层会是结实的，就像一根管子。）这一层成为导线对的第二根“电线”。由于中心的电线是在外层的中心（或轴上），因此这两根电线是同轴的。与屏蔽型双绞线相比，同轴电缆向外辐射自身信号的可能性更小，同时也更不容易摄取外部的信号。然而，同轴电缆要更为昂贵，故而仅限于一些特殊的用途。进一步说，同轴电缆正是有线电视所采用的电线类型。在建筑物内所使用的有线电视同轴电缆的粗细跟铅笔差不多，与非屏蔽型双绞线电缆相比显得有些不太灵活。

15.7.2 光纤规格

[352]

光纤几乎完全没有向外辐射信号的问题。不过，它也有点贵，价格大约是铜线电缆的两倍。但是，鉴于光纤几乎不会发生错误，所以得到了广泛的使用。另外值得一提的是，光纤可以在相当长的距离上传输数据。实际上，在 2001 年，就有一家供应商实实在在地演示了通过一根没有中继器的光纤实现跨美国大陆的传输。光纤具有非常高的带宽性能，因此对于需要处理大量数据的场合，用于传输每位的价格可能会非常低。绝大多数的新的广域网线路采用了光纤，同时，光纤在建筑物或大院内的局域网骨干网上也是很常见的。

当电话运营商最初接通光纤链路时，它们运行得非常好。这一方面就体现在，其间发生的错误率非常低。每根光纤只是由于接收者电路的物理限制而定格在大约 5 Gbps（千兆位/秒）的数据传输速率上。然而，当光纤使用了一段时间之后，有关工程师便意识到了一种简单、廉价并且可靠的光学处理途径，那就是通过在每一端都使用一个棱镜来实现多组信号在同一根光纤上的组合。这种技术被称为波分复用（Wavelength Division Multiplexing, WDM），有时也称为密集波分复用（Dense Wavelength Division Multiplexing, DWDM）技术。为此，所安装的每根光纤现在就可以承载 64~128 倍于最初想法的那么多的数据。由于线路用地成本及光纤安装本身的成本是一项最主要的因素，这便意味着后来的广域带宽成本的急剧下降。这种下降具体还表现为最近几年的长期电话费率的迅速下降。事实上，在许多情况下，如果相关客户同意购买本地服务，当地电话公司是可以承受得起为他们的客户免费提供长途线路的。

15.7.3 无线网络

如前所述，在数字数据传输领域中相对比较新的方式是通过无线介质（主要是数字无线电）来进行通信^①。这项技术显然适用于笔记本电脑和手持式计算机，但同时也适用于主机必须频繁地搬来搬去或者因物理条件限制而无法直接布线的场合。如果愿意，移动系统——机器人将会是另外一个很有前途的领域。关于无线通信的电子电气工程师协会（IEEE）标准是 802.11，其对应的市场营销名称 Wi-Fi（Wireless Fidelity，无线保真）为人们所熟知。网桥、路由器、个人计算机卡之网卡、外围部件互联标准网卡等各种设备都可以很方便地连接

① 其实，以前也曾经使用过无线网络。它们在传统上曾被用于军事领域或者诸如夏威夷群岛之类的铺设电缆成本极其昂贵、令人望而却步的地方。

到无线局域网上。相关设备可能会继续填充这一有利可图的市场。另一种无线协议——蓝牙 (Bluetooth), 则通过一种安全的、低成本的无线链路方便了诸如个人数字助理 (Personal Digital Assistant, PDA)、移动电话、笔记本电脑、计算机、打印机以及数码相机 (digital cameras) 之类的无线设备之间的信息交换。蓝牙协议正在由电子电气工程师协会审定为标准 802.15。该标准系列中的协议变种是为较短距离的通信而设计的, 这类网络有时被称为个人区域网络 (Personal Area Network, PAN) 或身体区域网络 (Body Area Network, BAN, 或简称体域网)。

无线方式非常容易受到来自外部源的干扰, 同时相关通信信号也很容易被其他设备无意或有意地截取到。无线方式的主要优点是两个通信设备之间不需要进行物理连接。由于存在噪声的问题, 无线通信便推动了比较强大的错误检测、校正及安全机制的构建。不过, 鉴于采用电缆和光纤的通信方式并不存在这些错误, 所以相关机制的开发稍有滞后往往也是可以接受的事情。

353

15.7.4 关于网络故障排查的说明

从实用角度来讲, 在排查网络问题时, 通常应当从检查物理层开始有关调查。物理层问题, 特别是间歇性问题, 可能会导致在其他各层出现形形色色的问题。因此, 往往应当首先通过验证和确认相应两个设备在物理层上的连接没有错误来开启网络故障的排查工作。

15.8 网络管理

15.8.1 简单管理工具

互联网控制消息协议 (Internet Control Message Protocol, ICMP) 和简单网络管理协议 (Simple Network Management Protocol, SNMP) 两种特殊协议与传输控制协议 / 网际协议簇一起运用于网络管理方面。互联网控制消息协议提供了多种功能, 不过对于网络管理人员而言最触手可及和容易理解的就是其为 ping 和 tracert (有时称为 traceroute) 等实用程序提供了基础。ping 例程是一种非常简单工具, 主要用于验证两台设备之间的连接。它向目标主机发送一条互联网控制消息协议回显命令。而目标主机则一般通过互联网控制消息协议回显应答来回复该回显命令。关于 ping 例程的各种选项支持发送大块数据、循环尝试 ping 操作等。通过测量响应时间及其变化情况还可以帮助网络操作人员识别网络中出现的性能问题。tracert 例程使用一连串的 ping 操作来发现连接两台网络主机的一系列的路由器。其可以给出相应信息传输路径上的每一跳的报告, 而这将有助于网络性能问题的进一步定位。

15.8.2 简单网络管理协议和网络设备管理

在历史上, 简单网络管理协议一直是网络管理软件用来与网络设备进行通信从而对相关设备实施监控、配置并为它们排除故障的协议。大多数可管理的网络设备往往拥有一组参数, 而这些参数将会提供相关信息或允许改变。进一步说, 这些参数将通过管理信息库 (Management Information Base, MIB) 来加以描述。从外部来看, 很容易认为有关设备实际存储了管理信息库本身。但实际上, 管理信息库只是一种用来描述相关数据及其语义和数据传输格式的方便的结构化的方式。有关设备可以采用任何便利的方式来存储相关数据值。互联网工程任务组已经对相当多的管理信息库进行了标准化, 其中包括关于诸如以太网端口的特定硬件类型的管理信息库以及关于诸如传输控制协议和用户数据报协议等各种协议的管理

信息库。不过，供应商们还另外添加了许多自己专有的管理信息库扩展。

354

管理员很少会直接看到简单网络管理协议。相反，像路由器之类的网络设备的管理信息库一般用于开发软件工具，以支持采用简单网络管理协议且通过图形化用户界面来实现对网络设备的远程管理。其中某些工具设计得非常精巧，所显示的图像上，既有灯光闪烁的各种设备，还有利用不同颜色来指示网络状态的地图。遗憾的是，这些软件工具往往都是专用的，所以许多大型的网络运营中心（Network Operation Center, NOC）常常充斥了几十台运行着各种不同软件包的工作站。这便要求在掌握各种软件包奇特功能的诸多操作人员之间展开广泛的换位培训和交叉训练。

当前的趋势是，在有关设备中放置上专用的特定的超文本标记语言（HyperText Markup Language, HTML）服务器实体，并采用基于应用层超文本传输协议（HyperText Transfer Protocol, HTTP）的网络浏览器来具体实施相关操作。这便意味着不再经常需要专用的管理软件，并且对交叉训练的需求也将有所减少。但是，我们仍然有必要了解相关网络设备的具体特性，而只是说，由于浏览器对所有这些设备都是标准的，故而所要求的培训工作大大减少了。

15.8.3 数据包捕获

对于以太网是使用集线器构建起来的情况，网络上的每个网卡都会看到通过局域网发送的每个数据包。通常情况下，网络适配器往往只会读取带有广播地址、多播地址或者网络适配器自身地址的数据包。但是，一些网络适配器也可以被设置为混杂模式（promiscuous mode），在这种情况下，它们就会读取网络上的每一个数据包。这便成为网络协议故障排查的有力工具。人们开发了许多非常精巧的工具，其中最著名的是由网络通用公司（Network General Corporation）开发的嗅探器软件 SnifferTM。此类工具常常拥有很多功能选项。例如，它们可以被设置为仅仅捕获满足特定标准的信息流、只有在发现某类触发事件之后才开启捕获过程、把所捕获到的数据包保存到硬盘上以及仅仅针对所关注的层级创建数据包的解码显示结果。遗憾的是，这类工具同时也有其不好的一面，因为如果所传输的信息没有加密，一些不道德的用户就可以使用有关捕获程序来查看特权信息和截获密码。

交换型以太网的发展在很大程度上解决了后面的这个问题，因为交换机只会把寻址到特定设备的信息流转发到与相应设备所连接的端口上。故此，数据包捕获机制就只能看到广播信息流、多播信息流以及发给捕获设备自身的信息流。当然，这意味着有关捕获技术无法再按照最初设想的那样实现有关意图。出于这些原因，用于大型环境的交换机通常具有一种称为端口镜像（port mirroring）的功能。这项功能将允许管理人员告知有关交换机获取所有来往特定端口的数据包并将其复制到另一个端口。然后，数据包捕获机制就可以嵌入相应镜像端口中，并且可以像以往那样捕获相关会话。需要指出的是，只有网络管理人员才能够打开和启用这项功能。

15.8.4 远程监控

由互联网工程任务组所定义的管理信息库，其中有一个管理信息库便涵盖了网络的远程监控（Remote MONitoring, RMON）。前往远程网络所在地进行故障排除和维护的成本往往会非常昂贵，因此最好能够通过网络远程诊断网络问题。鉴于路由器已经对其转发的每一个数据包进行了检查，所以它们正好处于能够完成这项功能的独一无二的位置上。关于远程监控的管理信息库定义了路由器中的远程监控代理可以维护的、基本路由器管理信息库中的计

数器之外的那些计数器。它们还可能包含完整的跟踪设施、基于应用层协议的统计信息以及其他有用的信息。

355

15.9 小结

在本章中，我们对网络系统的基本组件进行了概述。我们首先阐明了有关目的和动机，即为什么网络的研究和学习对于整个计算机系统特别是操作系统的理解是非常重要的。我们陈述了一些基本概念，描述了一个即将用于本章其余部分的网络模型，从而为计算机网络的进一步讨论奠定了基础。我们讨论了几个应用层协议以及传输层和网络层所使用的最知名的协议，即 TCP/IP 协议簇。我们还讨论了 IBM 公司及其系统网络体系结构协议所发挥的持续重要的作用。我们讨论了数据链路层，重点是以以太网，同时还讨论了令牌环和光纤分布式数据接口，并将它们与以太网进行了比较。我们讨论了共享式以太网的缺点，并说明了各种速率的交换型以太网已经开始主导局域网架构的主要理由。我们也涵盖了广域网以及一些并不常见的广域网协议，以及为什么某些情况下会使用有关协议的原因。接下来则介绍了物理层及其中的一些选项。最后，我们阐述了网络管理的相关内容，包括简单的实用程序、简单网络管理协议、通常的网络管理操作以及向基于超文本传输协议和浏览器的网络管理模式迁移的发展趋势。另外，我们也对远程监控协议进行了扼要介绍。

参考文献

- Abramson, N., "The ALOHA System—Another Alternative for Computer Communications," *Proceedings, Fall Joint Computer Conference*, 1970.
- ANSI/IEEE Standard, *Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, Std. 802.3-1985, May 1988.
- ANSI/IEEE Standard, *Token Ring Access Method and Physical Layer Specification*, Std. 802.5-1985, December 1987.
- ANSI/IEEE Standard, *Token-Passing Bus Access Method and Physical Layer Specification*, Std. 802.4-1985, March 1986.
- ATM Forum, *LAN Emulation Over ATM LNNI Specification Version 2.0 (AF-LANE-0112.000)*, February 1999.
- Beck, M., et al., *Linux Kernel Programming*, 3rd ed., Reading, MA: Addison-Wesley, 2002.
- Bertsekas, D., and R. Gallager, *Data Networks*. Englewood Cliffs, NJ: Prentice Hall, 1987.
- Comer, D., *Internetworking with TCP/IP Principles, Protocols, and Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1988.
- Martin, J., and K. K. Chapman, *SNA: IBM's Networking Solution*. Englewood Cliffs, NJ: Prentice Hall, 1987.
- Martin, J., and K. K. Chapman, *Local Area Networks Architectures and Implementations*. Englewood Cliffs, NJ: Prentice Hall, 1989.
- McQuillan, J. M., I. Richer, and E. Rosen, "The New Routing Algorithm for the ARPANET," *IEEE Transactions on Communications*, Vol. COM-28, May 1980, pp. 711–719.
- Metcalf, R., and D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM*, Vol. 19, No. 7, July 1976.
- Perlman, R., *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*, 2nd ed., Reading, MA: Addison-Wesley, 1999.
- Postel, J. B., C. A. Sunshine, and D. Cihem, "The ARPA Internet Protocol," *Computer Networks*, 1981.
- Stallings, W., *ISDN: An Introduction*. New York: Macmillan, 1989.
- Voydock, V. L., and S. T. Kent, "Security Mechanisms in High-Level Network Protocols," *Computing Surveys*, Vol. 15, No. 2, June 1983, pp. 135–171.
- Zimmerman, H., "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, Vol. COM-28, No. 4, April 1980, pp. 425–432.

356

网上资源

<http://www.bluetooth.com/bluetooth/> (商业产品)

<https://www.bluetooth.org> (标准组织)

<http://www.ietf.org> (互联网工程任务组; 审订了所有要求评定的规范, 包括网际协议、传输控制协议、用户数据报协议、网络地址转换协议、路由信息协议、第二版路由信息协议、点对点协议、第6版网际协议、无

类别域间路由协议、串行线路接口协议)

<http://www.ipmplsforum.org> (“互联网协议及多协议标签交换”论坛——延续了“异步传输模式”论坛)

<http://www.w3.org/Protocols> (主要是关于超文本传输协议的)

习题

- 15.1 计算机联网最初的主要动机是什么?
- 15.2 什么最终成为计算机联网的最显著的好处?
- 15.3 在我们所讨论的网络模型中,每一层均由每台计算机中的一个实体来作为代表。于是,每个这样的实体将会与每个与其相连接的另一个实体进行对话。该实体会跟什么样的其他实体进行对话呢?
- 下一更高层级的实体
 - 下一较低层级的实体
 - 在另一个系统中的对等实体
 - 以上都不是
- 15.4 在广域网中,就中心站点到达所有结点的速度而言,哪种拓扑结构是最高效的?
- 线形结构
 - 树形结构
 - 星形结构
 - 环形结构
 - 以上所有拓扑结构的通信速度都相同
- 15.5 哪种广域网拓扑结构在线路成本方面是最昂贵的?
- 星形结构
 - 环形结构
 - 部分互连的网状结构
 - 完全互连的网状结构
 - 以上所有拓扑结构都具有相同的线路成本
- 15.6 一些共享型局域网拓扑结构并不是非常高效——值得注意的是,共享式以太网的运行很少会超过 60% 的效率。是什么样的重大进展支持这样的局域网运行在较高的吞吐量级别上的?
- 15.7 动态主机配置协议是用来完成什么任务的?
- 把网际协议地址转换为介质访问控制地址
 - 把名称转换为网际协议地址
 - 获取网际协议地址及其他信息
 - 更新万维网服务器主机上的页面
 - 以上均没有正确描述动态主机配置协议的作用
- 15.8 把我们从迁移到第 6 版网际协议的悬崖边上解救下来的主要因素是什么?
- 动态主机配置协议
 - 网络地址转换协议
 - 无类别域间路由协议
 - 域名服务协议
 - 以上各种协议对我们延迟使用第 6 版网际协议都没有帮助
- 15.9 每个协议层都必须在对应报头中拥有一些信息,以告知接收者实体应该把传入的协议数据单元 (Protocol Data Unit, PDU) 传送给一个什么样的实体。在传输层报头中的什么信息将会告知传输控制协议或用户数据报协议,从而把相应数据包传递给哪个应用程序呢?
- 15.10 什么协议用来把网际协议地址转换为介质访问控制地址呢?
- 地址解析协议
 - 网络地址转换协议
 - 动态主机配置协议

- d. 内部网关路由协议
 - e. 上述协议均不涉及从网际协议地址到介质访问控制地址的映射
- 15.11 哪种物理介质具有最佳的抗干扰能力?
- a. 同轴电缆
 - b. 屏蔽型双绞线
 - c. 无线网络
 - d. 光纤
 - e. 所有上述介质均具有相同的抗噪声能力
- 15.12 根据我们曾经提到的内容, 局域网和广域网的区别主要体现在什么方面呢?
- 15.13 我们说过, 文件传输协议与我们讨论的其他两种协议相比, 具有某些特殊的地方。请举例说明。
- 15.14 即将用于管理大多数网络设备(尤其是那些廉价网络设备)的机制是什么?
- 15.15 在对网络进行故障排查时, 应当首先检查哪一层?
- 15.16 我们说用户数据报协议是一种“不可靠”的协议。但为什么我们不设计一种“可靠”的协议呢?
- 15.17 以太网协议是目前使用的唯一的局域网介质访问控制协议。这是否正确?
- 15.18 Cat 5 是什么意思?
- 15.19 嗅探器(SNIFFER)是什么?
- a. 炸弹检测装置
 - b. 用于分析网络协议的专用设备
 - c. 沉迷于吸入挥发性化学品的人
 - d. 用于窃取密码的软件程序
 - e. 以上各项均未给出嗅探器的正确描述
- 15.20 当我们使用 PING 命令并从主机获得响应时, 我们便一下子获得了许多信息。假设我们原本不了解任何情况, 那么我们刚才可能获取到了哪些信息呢?
- 15.21 域名服务协议是用来做什么的?
- a. 把网际协议地址转换为介质访问控制地址
 - b. 将名称转换为网际协议地址
 - c. 获取网际协议地址及其他信息
 - d. 更新万维网服务器主机上的页面
 - e. 以上都不是
- 15.22 网际协议地址通常表示为譬如“129.107.56.23”这样的方式。此类地址拥有两个部分, 其中, “129.107”是一部分, 而“56.23”是另一部分。这些部分是如何使用的?
- 15.23 为网际协议信息流选择路径的路由器也可能使用称为路由信息协议的协议。这个协议是用来做什么的?
- 15.24 电子邮件使用两种不同类型的协议, 例如简单邮件传输协议和第三版邮局协议。这两个协议有什么区别?

保护和安全

关于计算机操作系统的安全问题，过去所提出的有建设性的思想很少。在大多数情况下，安全性是通过控制物理访问来提供的。计算机就是被锁在一个带有大量空调的屋子里的庞然大物。允许访问系统的用户会允许访问任何文件及任何运行着的程序。伴随时间的推移，相关情况发生了变化。鉴于在同一时间通常运行着许多程序，而它们分别代表可能存在着竞争利益关系的很多用户，于是分时（time sharing）便掀起了这场巨大的变革。今天，系统的共享式访问已十分常见，尤其是在我们自己的家里。甚至当系统没有被共享的时候，它们也经常是连到了网络上。在许多情况下，即使是在家里，计算机系统也会连到局域网中。起码来说，许多机器可以通过拨号连接方式连到互联网上。尽管这可能只是间歇性的连接，但这种连接将会把我们的系统暴露在整个互联网世界中——一个威胁与惊奇并存的地方。

在第 16.1 节中，我们讨论了一些安全问题的起源，随后，我们把它们分解成若干不同的类别，进而描述了操作系统处理相关问题所需的机制，而其中的某些机制则需要驻留在操作系统自身之外的其他地方。接下来，在第 16.2 节中，我们对操作系统需要为用户提供的保护服务（主要是提供对文件的保密性）进行了概要介绍，并对这些服务总体的设计思路进行了阐述。除了操作系统必须为用户提供的服务之外，还需要给进程提供一种不同级别的服务。我们还在运行进程与操作系统之间建立起了重要的屏障，以保护操作系统和所有运行进程。在第 16.3 节中，我们会继续审视尝试通信和相互协作的进程所需要的一些服务。在第 16.4 节中，我们介绍了一般的网络尤其是因特网相关的安全，包括加密、认证和摘要，同时也讨论到了网络安全以及个人操作系统之外所发现的保护等相关话题。在第 16.5 节中，我们介绍了在网络和操作系统的安全管理中所出现的问题。在第 16.6 节中，我们对本章进行了归纳总结。

[359]

16.1 问题和威胁

我们应当有许多理由不去盲目地信任所有的程序。程序员可能会厌倦无聊、疲惫不堪，抑或懒惰、粗心、无知、愚钝，还可能存心不良或邪恶透顶。在某些情况下，其中的任意一项或者全部合到一起，都可以创造出一个能够破坏我们的工作甚至我们的系统的程序来。一般来说，我们需要担心两类人。黑客（hacker）对于家庭用户来讲非常危险，因为他们往往会攻击系统的薄弱环节，而这经常是大多数家庭用户甚至没有足够知识可以认识到或很少去处理的方面。同时，黑客也是最臭名昭著的，但他们只是问题的一部分。另一个不太出名的问题是，合法进入系统的人的未经授权的访问。此类内部问题涉及面非常广泛，具体包括发送辱骂性或威胁性的电子邮件、从账户偷盗钱财或者从库存窃取货物、浪费时间去浏览与工作不相关的网站或者在计算机上玩游戏、窥探其他员工的个人信息、抄袭其他同学的课题或论文，以及贪污受贿、敲诈勒索、把公司秘密卖给竞争对手，等等。大多数此类问题都是很难发现或控制的，因为参与这些行动的人可以合法地进入系统。一般来说，我们必须要通过

操作系统控制之外的某些手段来识别这些行为。我们需要依赖实地清点、审计等各种方法。从某些方面来看,黑客往往被迫使用一套有限的非法机制来获得访问权限,所以他们更容易控制。如果我们足够勤勉,我们可能最终是能够识别和保护大多数相关机制的。但是,在这之前,鉴于系统的复杂性,保护大型系统是非常困难的(也有人说是不可可能的)。

黑客通常会利用操作系统中的某个问题,设法去执行一个其并未被许可执行的操作。这往往会使他们以超级用户(supervisor)或管理员(administrator)所拥有的权限获准使用系统,而相关权限基本上将允许他们可以做他们想做的任何事情。大多数情况下,这些机制会利用到操作系统里的某个缺陷或漏洞。通常,操作系统供应商将会很快认识到这些漏洞,并发布相应的操作系统补丁,以切断黑客可以利用相关漏洞的途径。遗憾的是,这些补丁并没有像一套完整的操作系统发行版那样进行过完备的测试,所以并不是所有的用户都愿意安装这些补丁包,而是让这些漏洞继续暴露着。这对于那些必须管理用来完成许多不同任务的许多不同系统的企业管理人员来说,尤其如此。尽管单独个体或许能够很快地判定一个漏洞修复程序会否引发一个问题,而一个企业管理人员则可能因为要对许多系统负责,所以可能不太愿意冒这样的风险。

我们可能看到的黑客威胁可以归类处理,从而使我们能够确定如何去处理这些威胁。其中,首要的就是我们现在称为**恶意软件(malware)**的通用类型。恶意软件是一个比较新的词,由若干子类聚合而成,具体包括病毒程序、蠕虫、特洛伊木马和间谍软件。

360

16.1.1 计算机病毒

计算机病毒(computer virus)是那些以类似于生物病毒将自己插入活细胞中的方式而把自己插入其他程序中的程序段。当包含有病毒的程序在计算机系统上运行时,病毒将会把自己插入辅助存储器上的其他程序中。对于病毒的编写者来说,最难的部分就是让用户去运行包含有病毒的程序。现在,这往往是通过把病毒程序附着到电子邮件上而使用户加以执行的方式来实现的。当大多数软件是以软盘来进行分发的时候,一种常见的技术就是感染软盘引导扇区中所找到的程序。如果系统在软件安装后重新启动(经常会要求这样操作),而上一轮的软盘并没有从驱动器中取下,那么往往会从软盘启动,于是病毒就会传播到硬盘上。接下来,该硬盘便会感染插入系统的每一块软盘。一旦这种病毒在集体环境中被释放出来,那么其就基本不可能彻底被根除掉,因为许多软盘放得到处都是,各式各样的桌子的抽屉里、公文包里、家里的衣柜顶部,等等。

今天,我们往往会拥有称为病毒扫描器的保护程序驻留在内存中,监视程序是否携带病毒、正在准备复制或者运行,进而防止相应复制或执行操作。这些扫描程序通过匹配已知的指令模式或者异常行为(例如,一系列的系统调用,或者试图去修改特定的系统文件或注册部分)进行扫描识别处理。遗憾的是,鉴于新型病毒不断地被创造出来,数据和行为模式的数据库必须随时更新。虽然程序本身常常是免费的,但是在试用期以后,数据库的维护更新则不再免费。为此,许多人不会费劲去运行扫描程序,或者不会付费购买升级,于是,很久以前就应该被消除的病毒仍然在继续传播。就像在生物世界一样,一些病毒只是让人讨厌,而一些则会造成灾难性的危害。与那些破坏性较小的病毒比起来,那些会瘫痪系统的病毒往往可能扩散得不会很厉害。系统崩溃会引起用户的关注,从而可能导致相应机器上的病毒的根除。但是,小幅的减速问题通常可能不会被注意,或者可能是被容忍了,因为解决相关问题或者成本太高或者会耗费太多的时间。

16.1.2 特洛伊木马

特洛伊木马 (Trojans, 或简称木马) 是指那些并非是它们看起来那样的程序。这个术语据说源自于特洛伊战争期间所使用的一项技术, 当一方佯装从战场上撤退时, 却留下了一个巨大的马的木制雕像, 其实马像里面藏着一队士兵。守城的军队将木马拉进城里, 并为信以为真的胜利举行了盛大的庆祝活动。到了晚上, 隐藏的士兵从木马中爬了出来, 打开城门, 放进假装撤退且业已返回的己方部队, 从而占领和洗劫了这座城池。木马程序看起来是某种功能的程序, 然而, 其或者完全在做着别的事情, 或者在做它们看起来所做的事情的同时还做着别的什么事情——用户没有注意到的某类事情。例如, 相应程序可能看起来是一个屏幕保护程序, 但同时还安插了一个进程, 而该进程将会窃听所有密码, 并把密码发送给另一个国家的某个网站。一般来说, 防范病毒所用的相同技术也会防范木马。

16.1.3 蠕虫

蠕虫 (worm) 是类似于病毒和木马的程序, 但也有点不同。它们不会感染其他程序, 也不会假装做某类事情。当一个蠕虫程序第一次运行时, 它只是试图将自己传送到其他机器上, 并欺骗操作系统去启动运行。接下来, 在那台机器上, 它又会尝试把自己发送到另外的机器上, 以此类推。1988 年, 康奈尔大学 (Cornell University) 的一名博士生编写和触发了一个称为互联网蠕虫 (Internet Worm) 的小程序。他的意图是, 这个程序将是隐形的, 即不会做任何看得见的事情, 它将被设计成把自己传播到尽可能多的计算机上而不留下任何痕迹。如果代码正确运行, 那么只会有唯一的一个进程运行在众多连接到互联网的计算机上。遗憾的是, 代码并没有按设想的那样工作。该蠕虫程序迅速地向四处传播和扩散开来, 而且一台受感染的机器经常会把蠕虫传回到先前传给它的那台机器上。结果是, 这些各自并不占用太多处理器时间的小型进程, 伴随越来越多的蠕虫进程在受感染的每台机器上启动起来, 蠕虫程序开始淹没和浸满了系统。大多数情况下, 在不到 90 分钟之后, 蠕虫程序将会使被感染的系统无法使用。没有人可以实际确定到底有多少台机器被这种蠕虫所感染, 但据估计, 大约涉及 6000 台左右。另外, 该蠕虫程序基本上让互联网瘫痪了大致一天左右的时间。幸运的是, 它只攻击运行 BSD UNIX 系统特定版本的 VAX 和 SUN 计算机。蠕虫也可以被病毒扫描程序检测出来并加以根除。

蠕虫不一定是破坏性的。蠕虫的最初开发是在 20 世纪 80 年代早期的 Xerox PARC 装置上。相关蠕虫程序主要用来在下班时间通过网络进行分布式处理、广播通信和软件分发等活动。

16.1.4 间谍软件

间谍软件 (Spyware) 是一种特殊类型的木马。此类程序要相对温和一些, 它们不会损坏自己运行所在的计算机或者任何用户数据。它们通常所做的也就是向一些不相关的网站报告本机的活动。在最良性的情况下, 这些报告的信息只是确定哪些网站在被访问, 从而帮助某些系统把广告投放到用户可能比较感兴趣的那些网站上。事实上, 这里存在一个灰色区域, 其中的活动可以被看作是有实际益处的。纯朴的用户常常被一些缺乏道德原则的广告商误导而安装上“屏幕保护程序”和“浏览器工具栏”, 其实它们就是含有此类间谍软件的特洛伊木马。至少来说, 相关用户并不经常会被告知, 系统正在安装这些其他的功能。在最坏的情况下, 音乐光盘的供应商会趁着他们的一张光盘在计算机上播放的时候安装间谍软件

包,或者税务软件的供应商在他们的软件被安装的时候通过误导用户尝试实施数字版权管理(Digital Rights Management, DRM)来启动间谍软件包的安装,并且所有这些安装过程都是在没有通知软件用户的情况下完成的。在这两种情况下,系统性能都会有所降低,而且新的安全漏洞将会被暴露到互联网上。更恶意的情况则是,间谍软件可能用来窃取网站甚至信用卡账户的密码。幸运的是,一些特殊的扫描软件能够识别并会删除大多数的间谍软件。一个间谍软件程序所造成的破坏并不常常是太过惨重的。但是,当在初级用户拥有的机器上第一次运行扫描软件的时候,查出成百上千的间谍软件则是稀疏平常的事情,而且这些系统在此等扫描负荷之下基本不能再干什么其他事情。在本书写作之时,间谍软件类检测程序已经开始与病毒类扫描软件进行合并,从而形成一种更加通用的恶意软件类扫描程序。

[362]

16.1.5 拒绝服务攻击

尽管刚才描述的互联网蠕虫的本意是良性的,遗憾的是,从其运行所造成的副效应来讲,它还是另外一类攻击即拒绝服务(Denial of Service, DoS)攻击的一个例子。这种蠕虫所招致的结果是授权用户无法访问机器,或者说是操作系统因为很少在其被推向的这种压力极限条件下进行测试所以表现欠佳和不能正常运转,称为拒绝服务。拒绝服务攻击往往是指一种有意作用下的结果,而不是一种副效应。此类攻击非常多见,我们在这里主要描述其中较为典型的两种案例。第一种称为死乒攻击(Ping of Death)。ping是网络管理员用来测试网络连接的一种特殊的命令。服务器在接收到ping命令后,将会对发送者做出反馈和回应,这样发送者就可以知道目标机器是可达的。为了让程序更为实用和有效,发送者可以发送一个较大的数据包,且可发送若干次,以确认平均的响应时间。对于所附的数据包,一般设定最大应为64KB,但是也有可能蓄意创建一个大小超过64KB的ping数据包。遗憾的是,不少操作系统都拥有一种ping操作实用例程将会试图接收这种数据包,并将之放入一个至多只有64KB大小的缓冲区中。如果接收到超过64KB的数据包,那么数据将会把非设计中的某些东西覆盖掉,而这常常会带来严重的后果。这种攻击不会对发送者带来任何好处,而只会对目标对象造成伤害。

另一种拒绝服务攻击称为洪泛攻击(SYN Flood)。传输控制协议(TCP)网络连接一般从“三次握手”开始。首先,连接的发起方发送一个包含SYN标志的数据包,以告知传输控制协议,其在启动一个连接且接收方应当分配一些缓冲区并复位连接相关的一些数据字段。接收方回复后,发送方会再发送另外一个数据包来完成连接。在洪泛攻击过程中,发送方会发送许许多多的启动式SYN数据包来不断创建新的连接,但是从不对接收方的回复加以响应,从而消耗掉了大量的内存资源。当足够数量的未完成的连接被打开后,接收系统便可能会陷于瘫痪,或者仅仅是无法接受其他的连接从而拒绝授权用户的服务请求。

这两类问题在所有当今的操作系统协议栈中都已经进行了修复,但是它们可能依然存在于那些不容易升级的比较老式的网络设备中。其他这样的问题也经常被发现,并最终会得到修复。如果我们不是接连不断地去开发新的协议,或许这些问题最终都会得到很好的解决。然而,其他类型的攻击并不是那么简单就可以预防的,一个例子就是采用僵尸(zombie)的协同式攻击。一个僵尸是指一台安全性已被破坏以至于一个远程用户可以随意运行未经授权程序的机器。假定有一大群僵尸计算机,那么一个恶意用户就可以在同一时间让它们同时运行某一特定的程序。由好几万僵尸系统组成的网络并不稀罕,因为好多用户对自己机器的安全都没有经验,而僵尸机器往往可能不会表现出那些很容易就可以检测出来的

[363] 症状。(僵尸机器有时候也被称为机器人，一大群的此类机器则被称为僵尸网络。)相关程序可能是由合法的请求组成的——此类请求有可能是向网站服务器请求生成一个特定的页面，也可能是一些伪造的但从协议角度来看是合法的请求，故而很难检测或者预防。例如，我们也许会将一个页面发送到网站服务器上，这通常只用于维护的情景，因此大多数的用户并没有得到授权但请求本身看起来却是合法的。如果数千台僵尸计算机可以在同一时间被利用，那么它们就可能使一台服务器超载从而使合法用户被拒绝访问。即便相关服务器没有超载，服务器的整个通信连接也可能已被填满，故而还是会导致拒绝服务。与此同时，因为没有任何单一的一台僵尸机器看起来处于某种负荷之下，所以系统的所有者们可能甚至没有注意到正在发生的任何事情。

16.1.6 缓冲区溢出

为了实施破坏活动，病毒或蠕虫往往需要设法欺骗系统基于管态 (supervisor mode，也称为管理程序模式) 去运行有关的恶意代码。病毒或蠕虫驾驭这项技艺的一种最常见的方法就是利用一种称为缓冲区溢出 (buffer overflow 或 buffer overrun) 的程序编码错误。我们所提到的死拼攻击就是属于此类的一个例子。当进程存储数据超出缓冲区末端时，就会发生缓冲区溢出。在这种情况下，多余的数据将会覆盖邻近的内存区域。缓冲区溢出可能会导致进程崩溃或者输出错误的信息。它们可能是由蓄意设计的输入而触发的，或者试图去运行恶意代码，或者只是通过改变数据来使程序以一种并非计划中的方式运行。缓冲区溢出是许多软件脆弱性的起因，同时也是许多恶意攻击的基础。边界检查 (bound checking) 可以预防缓冲区溢出。然而，程序设计人员通常并不会考虑这些问题，而只是天真地假设有关的输入数据都是正确的。编译器可以生成始终执行边界检查的代码，但是程序设计人员出于效率的考量往往会关闭此类编译选项。

在下面的例子中，X 是当程序开始执行时内存中的某项数据，而 Y 则恰巧在它的毗邻区域。二者目前都是 0。

X	X	X	X	X	X	X	Y	Y
00	00	00	00	00	00	00	00	00

如果程序试图把字符串 “too much” 存储到 X 中，则其将会导致缓冲区 (X) 溢出，并破坏掉变量 Y 的取值。

X	X	X	X	X	X	X	Y	Y
‘t’	‘o’	‘o’	‘ ’	‘m’	‘u’	‘c’	‘h’	00

虽然程序设计人员根本就没有打算改变 Y 的取值，但是，Y 的取值现在已经由相关字符串的一部分所形成的一个数替代掉了。在这个例子中，即采用 ASCII (American Standard Code for Information Interchange，美国信息交换标准代码) 编码的大端字节序的 (big-endian，即高位优先的) 系统中，“h” 跟上一个内容为零的字节将会是数 26624 (译者注：字符 “h” 的 ASCII 码为 0x68，于是二者合到一起就是十六进制数 0x6800，对应十进制数即 26624)。因此，写入一个很长的字符串将可能导致诸如分段故障、进程崩溃之类的错误。

用来开发缓冲区溢出的相关方法根据计算机体系结构、操作系统和存储区域的不同而有

所变化。除了用于改变不相关变量的取值，缓冲区溢出经常被攻击者用来欺诈程序去执行来自蓄意设计的输入所导向的那些随心所欲而设计的恶意代码。攻击者用来获取控制的这项技术依赖于缓冲区所处的内存区域。例如，缓冲区可能是在堆栈区域，有关数据会先压到堆栈上，然后再弹出来。不过，也存在利用堆溢出（heap overflow）的情况。

通常情况下，当一个函数开始执行时，局部变量被压到堆栈上，并且只有在执行该函数的过程中才能被访问。这便存在一个问题，因为在大多数系统中，堆栈还会保存返回地址，即在当前函数被调用之前程序正在执行的位置。当该函数结束时，执行会跳转到返回地址。如果返回地址被一个缓冲区溢出所覆盖，其将会指向其他位置。对于第一个例子里所出现的偶发性缓冲区溢出情况，其几乎肯定会是一个无效的位置，即不会包含任何程序指令，故而对应进程将会崩溃。但是，通过仔细地分析研究一个系统中的代码，攻击者就可以巧妙地规划相关执行流程，从而使系统开始执行有关的攻击代码。当前，现代的操作系统已开始通过在逻辑地址空间随机分布代码和数据位置的方法来增加实施此类攻击行为的难度。

16.1.7 脚本和小应用程序

另一类型的恶意软件变种有时会在恶意网站中找到。一些机制不断演化发展，使得一个网站可以向客户端系统发送程序。这些机制包括脚本语言，譬如 JavaScript、VBScript，以及旨在软件虚拟机上运行的小应用程序（applet，有时也简称为小程序），相关虚拟机如太阳微系统公司（Sun Microsystems）的 Java 虚拟机（Java virtual machine, JVM）和微软公司的通用语言运行库（Common Language Runtime, CLR）。这两种机制通常需要以一种称为沙箱（sandbox）的方式在浏览器内部执行程序，如图 16-1 所示。这便意味着相关程序的活动是受到限制的，故而它们本来应该不会造成危害。例如，通常在浏览器中运行的程序是不允许访问本地磁盘驱动器的。然而，大多数的浏览器都允许用户去撤销相关限制，从而使一个可信程序可以去做一些事情，而这样的事情我们可能不会让一个我们并不放心的程序去做。如果我们是从我自己的公司网站得到的小应用程序，我们很可能就会信任它们。而如果小应用程序来自于其他地方，我们可能就不会信任。

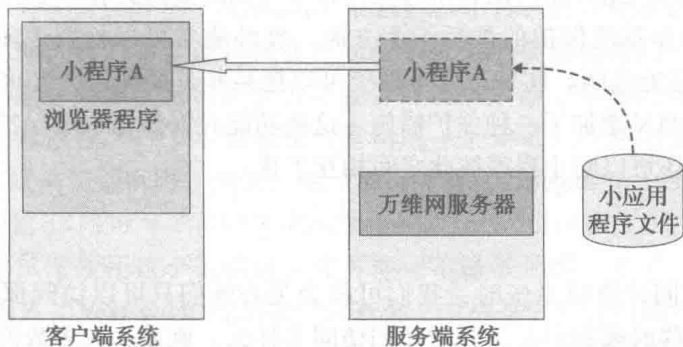


图 16-1 沙箱执行模型

小型文本文件（cookie）是另外一种被普遍认为会引发安全问题的机制。根据设计，网站服务器是无状态的——它们并不会保留关于各客户端的任何信息。（在网站服务器上运行的应用程序会这样做，但这并不是服务器本身的一项功能或特性。）这种无状态的性质限制了服务器可以完成的事情。为了扩展相关功能，浏览器进行了功能增强以允许一个服务器记录关于在浏览器系统上的一个网站的信息。这些都是比较短小的文本字符串。当一个浏览器

请求页面时，服务器就可以读取这些小型文本文件。它们将有助于使服务器看起来好像是有状态的。它们可以存储客户编码、所提出的最后一个问题、所访问的最后一个页面、购物车信息，等等。当你在某网站的页面之间进行跳转切换时，它们可以用来暂时性告知该网站你已经登录过了。需要指出的是，人们常常会认为这些小型文本文件可能包含有病毒或其他恶意软件，但这仅仅是一种错觉或误解。其间，跟踪式小型文本文件（tracking cookie）可能出于广告目的而构造造成是跨越多个网站进行信息共享的，但是它们无法、也不可能去损害一台计算机或者计算机上的其他信息或程序。

16.2 操作系统保护

16.2.1 保护

在前面的章节中，我们曾从若干不同的角度讨论过操作系统保护，但是在这里有必要再次提到相关内容。其中的一个例子如用于运行操作系统的管理程序模式（supervisor mode，也称为管态）和用于运行应用程序的用户模式（user mode，也称为目态）之间的分离机制。这种分离机制允许操作系统监控各种类型的操作，以确保它们不会去做任何引发灾难的事情。例如，操作系统有可能要确认一项输入/输出请求不会覆盖文件系统的元数据部分。然而，在许多情况下，这并不能防止某些滥用行为，例如，某个用户可能会让系统去删除自己希望删除的某个文件。不过，一些这样的滥用可以通过我们将会在下面展开进一步讨论的文件系统保护机制来加以减缓。

类似地，操作系统通过启用一个计时器（timer）来保护处理器的使用，并且防止用户程序改变该计时器。这将允许操作系统中止任何陷入死循环的程序，而不管其是否是有意这样设计的。在批处理系统中，当启动一道程序时，往往会给出一个相应估算的运行时间，并且如果该程序运行时间超过对应运行时间估值一定百分比，那么将会中止该程序。在交互式系统中，相关情况往往假定用户最终会中止操作，并且或者重新尝试，或者去做其他事情。无论哪一种情况，一个流氓程序（rogue program）都不能完全阻塞相应系统，因为操作系统可能是多任务的，同时，用户还可以利用与操作系统之间的交互终止一个死循环程序。

内存保护是操作系统保护的最后一个方面。硬件通常可以检查逻辑地址空间（logical address space）的寻址范围。此外，许多系统可以把某些页面标记为允许只读访问的或者是只执行访问的，从而又增加了一种保护措施。这些功能允许操作系统保护自身不受任何用户程序的侵犯，同时还可以防止程序彼此之间相互干扰。

366

16.2.2 认证

当用户远程访问计算机系统时，我们可能会关心他们只可以访问那些我们想让他们访问的资源，尽管有些时候我们并不介意他们访问了什么。例如，大多数的网站服务器，基本上都会允许任何用户去访问任何页面。然而，通过网站接口进行访问的网上银行系统肯定不想让任何用户都可以访问服务器上的所有信息。我们希望用户只可以访问他们被授权访问的账户。实际上，用来控制用户可以访问系统上哪些东西的相关机制包括两个部分，即认证（authentication）与授权（authorization）。认证是去验证通信的一方的身份。在用户访问银行账户的案例中，我们需要验证双方的身份。银行希望对用户进行身份认证，这是显而易见的。然而，直到现在，相反方向的身份认证——也就是说，在使用网上银行系统时，我们也

想要对银行进行认证——却并不那么明显。一类新的计算机欺诈已经出现，被称为**网络钓鱼**（phishing）。在大多数情况下，网络钓鱼诈骗都是通过电子邮件发送的。它试图引导用户去访问一个网站，而这个网站被伪装成譬如你的银行之类的网站。然后，它会诱使你输入一些机密信息，如你的信用卡号码或者银行账户号码，以及用户名和密码。它会使用一些看似合理的冠冕堂皇的理由来向你索要这些信息，比如，常常会声称需要对你的身份进行认证，而且通常会强调，是因为系统安全在某些方面遭受了破坏，所以系统才需要这些信息。鉴于此，我们现在已经明白，与向主机系统验证客户端的身份一样，向客户端进行主机系统的身份验证也是十分重要的。

身份认证一般会采取如下三种形式中的一种：你所持有的、你所知道的或者你所特有的（即你的个人特征）。关于你所持有的，具体例子如你的银行卡或者你的家门钥匙，但是卡和钥匙都有可能被偷走。关于你所知道的，具体例子如你的登录账号和密码，或者你的账户号码，但如果你在输入时被第三方看见或者如果第三方从通信线路上读取到相关信息，这些信息就有可能被截获和泄露。关于你所特有的，具体可能是声纹（voiceprint）、指纹（thumbprint）或者视网膜扫描（retinal scan），此类系统现在刚刚投入使用，从理论上讲，一旦它们得到进一步的发展，系统将会更难以欺骗从而认证结果会更为可信。同时采用两种不同的认证方法称为**双因素身份认证**（two-factor authentication，或称为双重身份认证），举例来说，当一个人使用银行卡时，还必须同时提供个人身份号码（Personal Identification Number，PIN）。

鉴于社会因素，密码是有问题的。最糟糕的密码是有时随软件或硬件而配备的默认密码。令人惊讶的是，通常情况下这些密码从来不做修改。而如果密码选择地不够小心，那么密码也很容易就会被猜到。这样的密码被认为是弱强度的。前面提到的互联网蠕虫曾使用了若干猜测密码的机制，并且取得了显著的成功。所以，我们建议采用**高强度密码**（strong password）。一般来说，如果密码包含有大写和小写字母、符号以及数字的组合，并且不包含任何名字、单词、重复符号或者序列（譬如123或tuv），那么这样的密码就被认为是高强度的。由名字或单词组成的密码往往通过猜测常见的单词或者与对应账户相关联的名字就可以破译出来，称之为**字典式攻击**（dictionary attack）。不过，高强度密码存在的问题是它们很难被记住。特别地，我们常常被建议在不同的系统中不要使用相同的密码，这便加剧了此类问题。于是，这些高强度密码经常可以在计算机显示器上粘贴的便笺上面发现，从而严重影响了它们的有效性。一个不错的技巧是构造一个句子，并使用每个单词的首字母即采用首字母缩写方式来构造密码。例如，由“World War 2 began in 1939!”（译者注：寓意第二次世界大战开始于1939年）可以生成一个密码“WW2bi1939!”。这是一个非常漂亮的高强度密码。当你试图使用这个密码时，你可能记不清楚是哪一年了，没关系！登录谷歌（Google）网站，输入“year wwii started.”以查询“二战是哪年开始的？”就可以了。现在，你的便笺上还可以写上一些仅对你自己才有意义的（即只有你才能看懂的）神秘暗示，比如说“W2”。

[367]

16.2.3 授权

一旦操作系统知道了用户是谁，下一项任务就是要确定对应用户可以执行什么样的操作。更具体地说，所允许的操作取决于正在访问的对象——对于一台计算机上的所有文件而言，我们通常不会拥有相同的权限。我们一直以来是把一个用户作为一个人来进行讨论

的，但是，在操作系统的上下文中，一个进程同样可以被认为是一个用户。决定一个用户可以针对一个对象（比如文件）执行什么操作的过程称为授权（authorization）。从理论上讲，操作系统可能拥有一个称为访问控制矩阵（Access Control Matrix, ACM）的数据表格。该矩阵的一个维度可以是用户，而另一个维度则是被操作的对象。矩阵单元格中的项则说明了允许对应用户针对相应对象可以实施的操作。图 16-2 展示了一个假想的访问控制矩阵。其中，我们可以看到各行标上了一些用户名，而各列则标以若干对象名。在这个例子里，我们可以看到有三个文件对象与一个打印机对象。温迪（Wendy）是美编部的设计人员，被授权使用激光打印机，但不能使用 gcc 编译器（GNU Compiler Collection, gcc，也称为 GNU 编译器套件）。安（Ann）和弗兰德（Fred）均可以阅读和编写自己的简历，而任何其他人员均不能阅读和写操作安和弗兰德的简历。同时，安和弗兰德都能够执行 gcc 编译器，但是他们都不可将 gcc 作为数据读入，也不可以写操作 gcc。注意，针对一个对象的可接受的操作集有可能与针对一种不同类型的另一个对象的可接受的操作集并不一样。例如，打印机可能支持停止队列的操作，而一个文件则不会支持停止队列的操作。

	安的 简历文件	弗雷德的 简历文件	gcc 编译器	激光 打印机
安	读/写	-	执行	-
弗雷德	-	读/写	执行	-
...				
温迪	-	-	-	写/ 停止队列/ 开始队列

图 16-2 访问控制矩阵示例

368

即便是从这一小部分的访问控制矩阵也可以清晰地看到，访问控制矩阵中大部分的单元格都是空的。试图保存整个访问控制矩阵将会浪费掉大量的内存。一台大型机器可能拥有成千上万的用户和无以计数的文件，并且大多数的用户只允许访问很少的一部分文件。鉴于这个原因，操作系统一般并不采用访问控制矩阵，而是会采用访问控制表或者权限表。访问控制表（Access Control List, ACL）附着在一个对象上，并且仅仅包含被授权针对该对象执行操作的用户。访问控制表的表项将会列出每个可以访问对应对象的特定用户以及他们针对这个对象可以执行的操作。图 16-3 给出了对应于图 16-2 中访问控制矩阵的一些访问控制表（译者注：确切地说，是三张访问控制表，分别附着在安的简历文件、gcc 编译器及激光打印机等三个对象上）。

	安的 简历文件	gcc 编译器	激光 打印机
安	读/写	执行	写/ 停止队列/ 开始队列
弗雷德		执行	

图 16-3 对应于图 16-2 中访问控制矩阵的一些访问控制表

另外，操作系统也可以采用权限表（Capability List, CL）。图 16-4 显示了与图 16-2 中访问控制矩阵相匹配的一些权限表（译者注：原文为 A CL，即一张权限表，应为笔误。确切地说，具体给出了两张权限表，分别对应于安及温迪两个用户）。权限表的表项给出了一个用户被授权操作的各个对象以及该用户针对各对象被授权执行的操作列表。

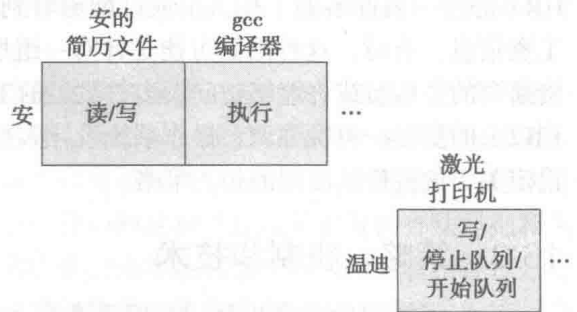


图 16-4 对应于图 16-2 中访问控制矩阵的一些权限表

尽管如此，按照我们已经展示的方式来创建这些表格中的表项，所创建的引用数还是会远远多于我们可能喜欢的数量。具体而言，以一所规模较大的综合性大学的学生访问一台计算机上的通用系统实用例程的权限设置的问题为例，不仅有成千上万甚至数万的学生，而且每个学期的学生都在发生变化——有的加入，而有的离开，无论以何种方式。为每一名学生设定所有必要的权限是一项重大的行政问题。然而，我们常常也会采用分组（group）或角色（role）的授权方式。我们可以创建一个称作“学生”的组别，并为该组分配针对相关必要对象的访问权限。在此基础上，当学生到来或者离开时，管理人员所需完成的事情就是把新的学生加入这个组里面以及把那些不再拥有访问权限的学生从组里移除到外面。赋予该组的访问权限可以由学生继承下来。角色授权方式是类似的，其允许我们在系统中拥有一个用户，并为这个用户分配一个角色。一个角色可能对应于特定项目组里的某个成员。该项目组成员所对应的所有用户应继承该项目组所共享的一组对象的一系列访问权限。当有成员离开这个项目组而进入另一个项目组时，管理人员所需完成的事情就是改变他们的角色，进而他们所有的访问权限也将随之发生改变。

369

一个必须要回答的问题是，授权应当在什么时候进行检查。一种选择是当一个对象被第一次访问（例如，一个文件被打开，一个套接字连接被创建，一个特定的打印机被假脱机系统所请求）时，就去检查相应授权情况。在此之后，最初的访问所包含的一组操作类型将不再进行另外的检查而直接准予执行。例如，如果我们打开了一个文件用于输入，那么这个文件将被允许读操作，但不允许写操作。根据操作系统所需的安全级别，我们可能会觉得这是不够的。我们可能会要求每一个独立的操作都应当对照相关访问控制表或权限表进行专门的检查——因为说不准在最初的访问过后，用户的相关权限可能已经被撤销了。例如，某用户即将被解雇，而他自己发现了这一情况，故而开始改写操作系统上的文件以试图消除其渎职的证据。虽然管理人员废除了该用户的密码，但是相关操作已经在进行中了。对于此类问题，我们将会安全级别相关主题中深入展开讨论。请注意，这是操作系统设计人员必须要决定的，即是否需要在操作系统中提供相应机制来支持有关的授权检查功能，同时，系统管理人员也必须要决定是否调用相关功能。对于百货公司而言，此类安全机制无论对系统本身还是对于维护行政成本来说都是不值得的。但是，对于银行而言，则完全是另外一码事。

另外，还有一个问题，就是授权应当在什么级别上实施。通常情况下，系统中的对象会组织成一个或多个树形结构，譬如，硬盘驱动器上的文件系统就是一个例子。我们往往授权一个用户可以访问一个特定的主目录（home directory）。这意味着，针对一个目录的访问权限将会自然延伸到其子目录上，除非相关权限被重新设定。一般而言，我们也可能会对一个目录下的任意文件的访问权限进行重新设定。在某些系统中，我们还可能针对数据库的各

个部分（有时候是针对一组字段）分别进行权限设定。为此，人力资源（Human Resource, HR）部的一名办事的工作人员或许能够看到所有员工的家庭联络信息，但却看不到他们的工资信息。有时，这种限制可能是针对一组取值（譬如一组记录）来展开的。于是，一名负责薪资的办事员或许能够访问大多数员工的工资记录，但其中不包括那些位阶在某一管理级别以上的员工。再次强调，操作系统设计人员必须决定是否提供相应机制，而管理人员必须设定关于这些机制使用的相关策略。

16.3 策略、机制和技术

有许多类型的安全机制，它们通常在大多数较大型的操作系统中都可以找到。在本节中，我们将会介绍其中的几种常见机制。但在我们考虑相关机制之前，首先制定相应的策略是非常重要的，也就是说应当毫不含糊地确定用户可以做什么、不可以做什么以及他们必须做什么。

370

16.3.1 安全与保护策略

任何试图成为安全的网络都必须拥有一系列明确说明如下事项的策略：

- 什么是准许用户做的。
- 什么是不允许用户做的。
- 什么是要求用户做的。
- 而如果相关规程未被遵循，相应的惩罚又会是什么。

即便是在家庭环境中，如果家长希望限制孩子访问某些类型的网站，那么他们应当清楚地说明是什么样的具体限制。防火墙机制并不完善，很有可能家庭中的孩子最终会比他们的父母更懂计算机和具有更强的计算机操控能力，故而能够成功跨越许多安全机制。为此，应当事先确定和声明，如果做了那些被禁止的事情将会受到怎样的惩罚。如果没有提前清楚地说明相关限制，即便是要解雇一个发送了威胁邮件的职员都是十分困难的。

同样，员工必须明确地被告知，就备份信息、某些环境采用加密措施、保护他们的计算机等事项而言，他们的具体职责是什么。如果让他们去运行病毒检查程序和防火墙并且保持软件更新，那么他们应当被提前告知，如果他们不这样做，将可能发生什么样的问题，而不遵守相关策略将会受到什么样的惩罚。

16.3.2 系统崩溃保护：备份

鉴于有时会发生系统崩溃，所以操作系统必须提供相应的处理机制，即系统崩溃保护（crash protection）。首先，运行的系统可能会出现崩溃。尽管操作系统同以前比较起来要好多了，但是没有任何一个重要的程序是真正调试过的。当系统崩溃时，我们必须能够从相关系统崩溃困境中恢复过来。我们已经讨论过一些处理相应崩溃的机制。事务（transaction）是其中的第一项机制。通常情况下，我们会采取一系列的关于文件或者数据库的更新操作，它们协同工作并可共同定义为一项事务。我们可能正在把一件昂贵的器材从一个仓库转移到另一个仓库。如果我们采用的是单独的文件或数据库，那么其中一项更新操作需要反映该器材离开了这个仓库，同时另一项更新操作需要反映该器材现在放在了另一个仓库中。如果系统是在其中的一项更新操作完成之前就崩溃了，那么我们要么将会失去该器材的所有线索，要么会以为我们所拥有的器材数量比我们实际所拥有的器材数量要多一个。通过把这些更新

操作捆绑成为单一的一项事务，系统就可以确保这两项更新操作要么全部完成，要么哪一项都不做。我们主要通过日志（logging）、检查点（checkpoint）和回滚（rollback）等方法来实现相关事务机制。

数据丢失的另一个可能的根源在于磁盘驱动器的物理崩溃。有时候，磁盘上的读写磁头会实实在在地划损盘片，从而擦掉盘片涂层。另外一些时候，我们也有可能遇到轴承或者电子故障。尽管有些时候从出现故障的磁盘驱动器上也有可能恢复一些甚至全部数据，但这件事情肯定是代价十分昂贵并且非常耗费时间的。一种应对这种可能性的更好的办法是把相关数据备份到可移动介质上。从历史上来看，因为磁带的成本比较低，所以这种备份往往是把相关数据复制到磁带上。对于小型系统，则常常使用软盘或者稍高容量的相关产品来进行备份。今天，个人计算机备份（backup）最常使用的是光盘（Compact Disk, CD）或数字化视频光盘（Digital Video Disk, DVD）。不仅磁盘驱动器会发生崩溃问题，用户有时候也会引发“崩溃”事件。每一个系统管理员已经习惯于听到有用户删除了一个的的确确所需要的文件，并且对应用户非常恳切地请求系统管理员帮忙恢复相关文件。因此，备份同样是值得去做的一件事情，因为文件可能会因为人为的错误或软件的问题而被删除或损坏。

[371]

我们可以采取许多种方法来备份系统，但是其中有一种方法最为可取。之所以这样断定，还牵涉如下的事实，即大多数的操作系统在文件目录中都拥有一个标记，用来指示自从上次备份以来对应文件是否发生过修改变化，常常把这个标记称为归档位（archive bit）。至于最佳的处理程序的具体细节，往往不尽相同，但为了解释说明，我们将假设我们想让系统备份能够非常容易地使用，所以每个周末我们都可以为整个系统建立一个备份。在备份时，将会清除所有文件的归档位，以说明我们已经拥有了相应文件的一个副本。鉴于系统每天都在运行，故而系统可以针对每个发生更改的文件设置其对应的归档位。于是，在每天结束时，我们可以运行另一个不同的备份程序，即其仅仅会复制当天发生过改变的文件，并且我们将会把日期标记到对应副本上。再次强调，我们每一天都这样做。于是，当有用户请求恢复已被删除的某个文件时，我们可以按照时间逆序的方式来搜索每日备份得到的所有的文件副本，直到我们找到相应的文件为止。在大型的集中式运营部门，其备份机制通常会具体记录哪些文件是在哪天的备份中，为此，甚至都没有必要在所有各天备份的范围内去查找所需的文件。另外，这类系统同时还存在以下几项关键特征。首先，备份应当与对应计算机不在同一个房间内。因为，万一发生火灾，相同房间里的备份将很可能也被毁掉。更为重要的是，基于同样的原因，本周以前的所有备份介质甚至应当不要保存在同一栋建筑物中。因为，如果发生了洪水，将会使整个建筑物在一段时间内都无法进入。而如果有异地的备份可以使用，那么数据可以在另外的一家机构的系统中实施恢复，并且相关恢复操作将会更为迅捷。

在数据象征着大量金钱的情况下，则可以考虑选用另外一种机制。具体而言，相关数据往往可能包含了工程或艺术方面的设计，也就是说，数据当中融入了创造性的成果，所以它们的价值是难以估量的。换句话说，这些数据可能是的的确确无法替代的。在这种情况下，我们可以采用动态备份系统，也就是说，每当有文件发生改变，就须把对应文件写入远程的备份机器里。显然，这种解决方法比较昂贵，而且通常会牵涉行政管理。然而，对于这样的场合来说，相关举措则是非常值得的，即便心境平静下来明明知道有关文件是安全的。总而言之，把存储介质进行异地存放无论如何是十分重要的。

如果文件是在笔记本电脑上或者用于备份的介质实体经常被带到单位外面，那么对文件或介质采用加密处理就是一个合理的意见。这样，即使计算机或备份被盗，相关数据也不会

泄露出去。

对于经常性的备份，可以选择使用廉价磁盘冗余阵列（RAID）的磁盘组织方式，我们曾经在第 14 章就此进行过讨论。一些廉价磁盘冗余阵列构型建立在相当低下的成本基础之上，却可以提供实实在在的高可靠性，并且还可以减少由于驱动器故障而导致的数据丢失的问题。不过，它们无法解决由于人为或软件错误而导致的文件丢失问题。

16.3.3 并发性保护

我们已经说过，我们构建操作系统，会在运行进程之间建立许多保护机制。我们还提及，我们希望在多个进程之上建立高层次的系统，而在多个进程之上建立系统需要进程间的通信能力。因此，我们需要相关机制创造便利条件以实现进程间的通信。其中，共享内存就是操作系统能够为这些应用程序提供的一种机制。于是，我们为对应进程提供了一种手段，规定了它们想要协作并且共享访问某部分内存。我们在第 9 章已经讨论了共享内存使用中可能会出现的一些潜在问题，并提到，利用操作系统必须同时提供的锁定机制就可以解决相应问题。然而，这便又带来了牵涉死锁的另外一个潜在的问题。在这种情况下，通过锁的设定和释放的操作次序的一致性则是能够避免死锁的。

16.3.4 文件保护

对于多用户系统，操作系统还必须提供使文件私有化的相应机制。私有化并不一定意味着只有唯一的一个用户可以访问某一给定的文件，系统还必须可能让多个用户去共享同一个文件。在第 6 章，我们曾讨论过旧版本的 UNIX 和 Linux 系统中所用的关于文件所有者、同组用户（组内成员由系统管理员定义）和所有其他用户的访问权限指定机制，且这些权限一般通过 `chmod` 实用例程来进行设置。在第 18 章中，我们将讨论在 Windows NT 操作系统家族中指定文件访问权限的对应机制。而在第 19 章中，我们还将介绍 Linux 系统中提供的更新的相关机制。

有时，并发进程之间的通信会涉及文件级别的信息共享。大多数的操作系统往往在缺省情况下会允许各个独立进程并发地访问文件。为了避免在一个或多个进程写操作一个文件时可能出现的问题，相关进程必须使用同样的锁定机制以及合理次序的上锁、开锁操作来实现文件使用的同步，就像我们在同步使用共享内存时所做的那样。此外，关于文件保护，我们还将 16.4.1 节展开进一步的讨论。

16.4 通信安全

通常情况下，运行在一个系统上的一个进程需要与在另一个系统上运行的进程进行通信。而当我们通过通信链路从一台计算机发送到另一台计算机时，在发送信息的层级上，可能会出现三种潜在的问题，如图 16-5 所示。安全系统中的通信往往表示为是在两者之间，不妨称之为艾丽丝（Alice）和鲍勃（Bob）。第一种问题是指，第三方可能读取（或截获）相关消息。第二种问题是指，第三方可能发送（或插入）伪造的消息。而最后一种问题是指，第三方可能篡改由授权用户所发送的消息。我们所关心的就是保护系统防御

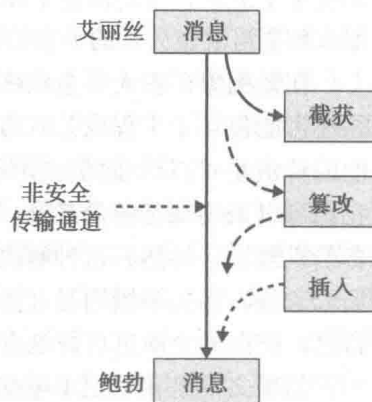


图 16-5 通信威胁

所有这些问题,即通信安全的问题。

我们通常使用的一类机制是由精心制作的关于如身份认证之类的特定功能的协议所组成的。例如,一种称为 Kerberos 的协议是针对身份认证而开发出来的。该协议被广泛使用,几乎已经成为一种事实上的标准。在 Windows 2000 系统中,默认的身份认证协议就是 Kerberos。设计这样的协议并使其足够安全,是一件非常复杂的事情,故而凭借其自身的头衔甚至就是一个专业。使用这样的安全协议可以使我们确定,我们正在与我们想要通信的一方进行通信,从而能够减轻让第三方把伪造信息混入通信流而不被发现的大部分的问题。

373

16.4.1 加密

有人则开发出另外的一类算法来确保相关的消息不会遭受到上面所列出的那三种问题中任何一种问题的影响。这些算法被用来加密系统间所传输的消息,从而使它们不能够轻易地被第三方所读取或读懂。如果消息连读都不可能,那么它们也就不会被更改了。因此,加密(encryption)手段可以消除或者至少可以减轻这三种问题中的两种问题。加密是指采用一种已知的算法把消息(通常称为明文(plaintext))打乱,相关算法往往会使用一个被称为密钥(key)的特殊的数。整理和解读所接收到的消息,将其展示为原先消息的过程称为解密(decryption),如图 16-6 所示。这些算法一般建立在这样的事实基础之上,也就是说,当第三方捕获到加密的消息时,其在不知道密钥的前提条件下试图去解密相应消息将会是不可计算的(computationally infeasible)。从理论上讲,拦截程序可能会尝试每一个可能的密钥值,称之为蛮力攻击(brute force attack,或称为穷举式攻击)。因此,短语“不可计算的”是指,运行相应算法尝试所有可能密钥将会耗费相当长的时间,从而使看到对应信息时其已不再具有传输时的价值。遗憾的是,不可计算的含义是在不断变化着的。我们都知道,大致每 18 个月处理器的速度就会翻一番,故而,5 年前的不可计算的东西放到现在或许已是非常容易的事情。

我们一直在讨论的加密主要是在消息传输背景下,但是,加密也可以用到文件系统中。万一计算机被盗或者丢失,这对于保护相关敏感信息就会非常有用了。随着系统变得越来越轻便,这可能已经成为一项十分有用的功能。无论是对于掌上电脑和手机,还是对于笔记本电脑,都是如此。它们是把文件存放在随机存取存储器(RAM)中,而不是辅助存储器(secondary storage)中,不过它们通常仍然可以进行加密。

对称密钥加密

有时候解密过程使用的是与加密过程相同的密钥。在这种情况下,密钥必须只有发送者和接收者才能够知道(但是,对于某一给定的加密消息也可能有许多接收者,那么所有这些接收者均须知道对应密钥)。采用这种密钥的算法被称为是对称(symmetric)密钥算法,有时也称为共享密钥(shared key)算法或者隐秘密钥(secret key)算法。图 16-7 展示了共享密钥系统的工作机理,其间,由艾丽丝和鲍勃所共享的隐秘密钥记作 $K_{A,B}$ 。

有若干种采用对称密钥的不同算法。其中,数据加密标准(Data Encryption Standard, DES)曾是使用了许多年的标准算法,但是该算法已不再被认为是安全的。在 2001 年,一种被称为高级加密标准(Advanced Encryption Standard, AES)的新算法被提出。数据加密标准使用的是 56 位的密钥,而高级加密标准根据用户需要可以使用 128 位、192 位或者 256 位的密钥。当高级加密标准发布时,采用一种成本低于 10 000 美元的专门的硬件系统基于蛮力攻击方式在几个小时之内就可以破译数据加密标准算法。而采用一台类似的但要快得多的机器来破译高级加密标准算法,则需要花上 149 万亿年。当艾丽丝和鲍勃彼此互不认识

时，采用这种共享密钥机制会带来一个问题，也就是说，他们往往不愿意交换密钥。解决此类问题的一种比较成熟的方法是利用可信的第三方（Trusted Third Party, TTP）生成一个密钥并发送给他们。这种解决方案要求用户双方均须真正信任这一可信的第三方，并且该可信的第三方应当永远在线和一直可以接受访问。现在，已经出现了一些新的协议，如 Diffie-Hellman（迪菲-赫尔曼密钥交换协议）和 RSA，允许两个用户动态地生成一对密钥，就像下一段即将讨论的那样，并通过一个不安全的网络来交换这对密钥。

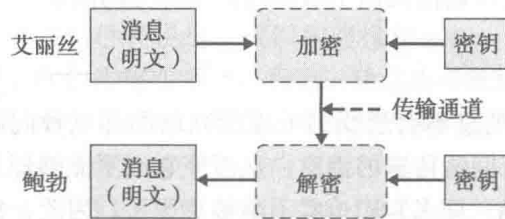


图 16-6 加密

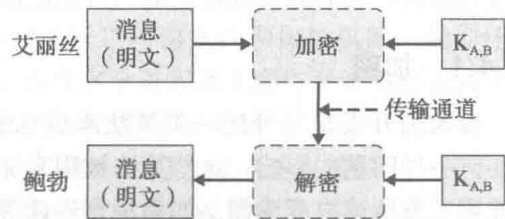


图 16-7 对称密钥加密

非对称密钥加密

其他的算法则一般使用共同生成的一对密钥，其中的一个密钥用于加密过程，而另一个密钥则用于解密过程。为此，这些算法通常被称为非对称密钥算法（asymmetric key algorithm）或者是公开密钥算法（public key algorithm）。关于这些算法，存在两个非常有趣的实情。第一个实情是，这对密钥中的一个密钥可以让整个世界都知道，称之为公钥（public key）。而另外的一个密钥则不公开，故而被称之为私钥（private key）。事实上，这往往就是实际情况。相关工作机理如图 16-8 所示，其中，鲍勃的公钥被标记为 K_B^+ ，而其私钥则被标记为 K_B^- 。

具体而言，如果艾丽丝想给鲍勃发送加密消息，她可以使用鲍勃的公钥来加密对应的消息。由于只有鲍勃才知道对应匹配的私钥，所以只有鲍勃能够读取相应的消息。非常有趣的是，每对密钥中无论用哪一个密钥来进行加密都无所谓，只要另一个密钥被用来解密就可以了。于是，鲍勃可以使用他的私钥来解密消息，然后发送给艾丽丝。如果艾丽丝确信她所拥有的鲍勃的公钥确实是属于鲍勃的，那么她就会知道对应消息的确是由鲍勃所发来的。（这里假设该消息可以另外通过对应协议来加以验证）。关于使用公钥加密的另外一个有趣的实情是，不同的密钥对可以按任何次序进行使用。因此，艾丽丝可以先使用鲍勃的公钥加密消息，然后再使用她自己的私钥进行对应消息的二次加密。而鲍勃可以按相反的次序或者相同的次序使用相应密钥来进行消息的解密处理。这种特性往往用在一些电子商务系统中。就像隐秘密钥加密一样，公开密钥加密也有若干种算法。多年来，其标准算法一直是 RSA 算法（或称为 Rivest, Shamir, Adleman 算法），是以此算法的开发人员的名字而命名的。该算法建立在素数（prime number，也称为质数）的基础之上，并依赖于如下的事实，即存在高效的算法用来验证某个数是否为素数，但却尚未知道有高效的算法可以找到某个数的素因子。

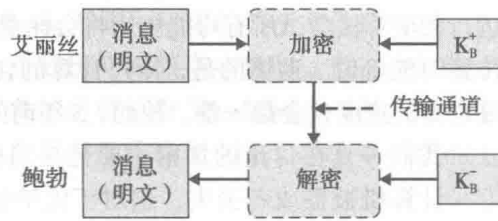


图 16-8 非对称密钥加密

374
375

16.4.2 消息摘要

在某些情况下，我们不一定要隐藏消息的内容。我们只是希望确保其没有被修改过。在

这种情况下，我们可以计算一个简单的、更为快捷的函数，称之为消息摘要（message digest）或者散列（hash）函数。在图 16-9 中我们简单地说明了这种函数及其使用过程。此类函数把一条长消息分割形成一系列较短的片断（一般约为 512 位），然后将它们与一个不能轻易逆转的所谓单向函数相结合，所得结果是一个固定长度的消息摘要——通常大约为 128 位。目前关于消息摘要流行的主要是两种算法，其一为产生一个 128 位散列码的 MD5 算法，另一种则是输出一个 160 位散列码的安全散列标准（Secure Hash Standard, SHA）算法。MD5 通常用来验证经由 HTTP 或 FTP 从互联网上下载得到的文件，特别是程序。文件下载时往往会伴随有文件的消息摘要，一般带有扩展名 .md5。针对下载得到的文件运行公开可用的实用例程，便可以计算出一个新的摘要。如果新的摘要与下载得到的摘要相匹配，那么就可以确信对应文件在上传到服务器之后没有发生过改变，同时下载过程中亦没有对该文件进行过更改。遗憾的是，现在已经发现，仅仅利用适度的计算能力就可以对 MD5 算法进行破译，因此，今天该算法只能主要在确保文件是否正确下载方面发挥作用了。

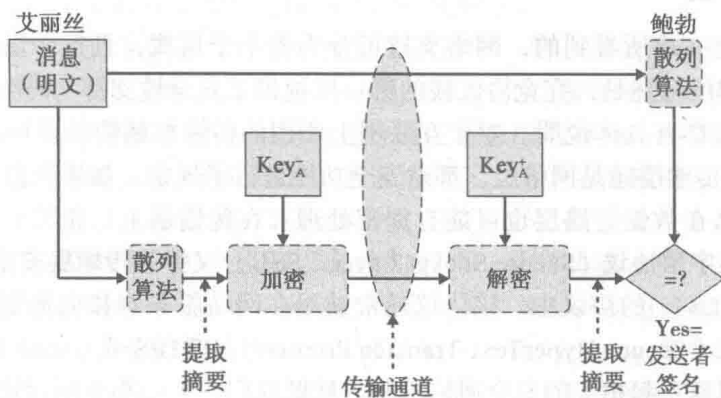


图 16-9 消息签名

16.4.3 消息签名与证书

通过将消息摘要与公开密钥加密机制相结合，艾丽丝就能够有效地对消息进行电子签名。设定艾丽丝准备发送消息 M ，于是其首先会创建该消息的摘要。随后，她用自己的私钥对该摘要进行加密，并把相应消息和经过加密的摘要同时发送给鲍勃。鉴于鲍勃是知道艾丽丝的公钥的，所以他可以解密这个摘要。接下来，鲍勃就可以针对所接收到的消息运行公开可用的摘要算法，并对其计算得到的摘要与先前解密得到的摘要进行比较。如果两者相等，那么他就可以知道（和可以证实）艾丽丝发送了对应消息。这样便可以保证艾丽丝以后无法否认（repudiate）该消息。请注意，为了在以后的日子里证实这一切，鲍勃必须保留对应消息、签名摘要，还有艾丽丝的公钥，因为艾丽丝有可能在后来会改变她的公钥。同时还应注意，鲍勃亦不能改变对应消息，因为其仍然需要举证是艾丽丝发送了该消息，因此这样也可以保护艾丽丝防止鲍勃篡改消息。

消息签名的一种特殊的用途是用来对客户机或者服务器进行身份验证，相关过程会产生一个用于验证身份的证书（certificate）。首先，运行一个特殊的程序以生成一份初始的证书。对于银行来说，其会在它们的服务器上完成此类事情。然后，这个初始的证书会发送到一家认证机构（Certificate Authority, CA），而该认证机构将使用自己的私钥对初始证书进行加密，并将加密结果作为成品证书（finished certificate）返回给请求实体。现在，银行就可以把这

份证书安装在它们的服务器上了。同时，银行还可以把这份证书发送给其客户以用来证明它自己的身份。因此，当有浏览器试图建立与这家银行的服务器之间的安全连接时，该服务器将会把对应成品证书发送给相应的浏览器。而该浏览器就可以通过使用上述认证机构的公钥来解密这份证书，从而验证银行的身份。一般来说，广为接受的认证机构的公钥往往都已内置在大多数的浏览器中。为此，浏览器使用相应认证机构的公钥对银行的证书解密后，用户接下来就可以知道其浏览器是真的连接到了银行的服务器上。

我们之前曾提到过，艾丽丝可以使用鲍勃的公钥来加密她要发送给鲍勃的消息。其间的一个问题是，艾丽丝必须确保这个密钥真的是鲍勃的公钥。她能够完成这项确认的方法就是，让鲍勃去利用一家认证机构基于该机构自己的私钥来签名他的公钥。这样，艾丽丝就可以使用该家认证机构的公钥去打开密钥，从而验证其中所包含的确实是鲍勃的公钥。采用这种方式进行电子签名的消息在法庭上从法律角度也是会接受和采纳的。

16.4.4 安全协议

正如我们在上一章所看到的，网络支持可分为若干个层次，且每一层都提供一定的功能。一个非常有趣的问题是，究竟协议栈的哪一层提供了安全性支持？碰巧的是，安全功能在几个不同的层次都有具体说明。对于互联网上所用的传输控制协议及网际协议（即 TCP/IP）来说，无论是传输层还是网络层，都对安全功能进行了规定。如果我们使用了 802.11 无线网络连接，那么在数据链路层也可能有加密处理。在传输层上，相关安全功能被定义在一个所谓安全套接字层协议（Secure Socket Layer, SSL；又称为传输层安全协议（Transport Layer Security, TLS））的协议中。该协议通常被用在网站服务器和浏览器之间，和称为安全超文本传输协议（Secure HyperText Transfer Protocol, HTTPS 或 secure HTTP）的应用层协议相结合，共同建立起相应的安全通信机制。就像我们在上一小节所讨论的那样，通过一份证书，对应服务器的身份得到了验证，从而保证对应客户进程是在与相应的服务器进行通话。这两个实体在一开始使用他们自己的公钥和私钥进行非对称加密。接下来，它们共同决定了一个临时的安全的会话密钥（session key），并采用这个密钥进行对称加密和继续相应的会话过程。与非对称密钥相比，对称密钥会更加有效，不过，对称密钥的重复使用会带来一定的风险，因此，对称密钥往往是针对单一的连接而生成，用完后就会丢弃掉。

在网络层也会提供安全支持，相应的协议被称为网际安全协议（IP security, IPsec）。网际安全协议是由互联网工程任务组（Internet Engineering Task Force, IETF）开发的一组协议，用来支持网际协议层的数据包的安全交换。其支持两种加密模式，即传输模式和隧道模式。传输模式只对消息内部的数据进行加密，而忽略了数据包头部。而隧道模式则会同时对数据包头部及消息进行加密，故而更为安全。网际安全协议对于发送方与接收方将使用共享的公钥，这些公钥是通过一种所谓的互联网安全联盟和密钥管理协议（Internet Security Association and Key Management Protocol/Oakley, ISAKMP/Oakley）来进行交换的，该协议允许接收方获得公钥，并采用数字证书方式对发送方的身份进行验证。与传输层安全协议相比，网际安全协议更为灵活，因为它可以与所有的互联网传输层的协议（包括传输控制协议 TCP、用户数据报协议 UDP、网际控制报文协议 ICMP）一起使用。不过，网际安全协议也要更为复杂些，并有处理方面的管理开销，因为它无法使用传输层上的那些安全增强功能。

在应用层上也有安全支持，相应的协议被称为恰到好处隐私协议（Pretty Good Privacy,

PGP)。恰到好处隐私协议采用的是公开密钥系统，其间每个用户都拥有一个公钥 / 私钥对。为了创建数字签名，恰到好处隐私协议会根据用户名及其他的签名数据生成散列码。随后，该散列码将使用发送方的私钥进行加密。而接收方则使用发送方的公钥来对散列码进行解密。如果该解码结果与作为消息的数字签名而发送来的散列码相匹配，那么接收方就可以确信对应消息是由所声明的发送方发送的，并且没有被更改过，无论是意外地改变还是有意地篡改。恰到好处隐私协议有两个版本，其中一个版本采用 RSA 协议交换会话密钥，而另一个版本则采用 Diffie-Hellman 协议。进一步说，RSA 版本使用 MD5 算法来生成散列码，而 Diffie-Hellman 版本则使用 SHA-1 算法来生成散列码。

16.4.5 网络保护

有几种实际上并不是在操作系统内部的工具，可以用在网络中来提高安全性。但是，我们还是要简单地介绍一下这些机制，因为它们会影响到操作系统内部的安全功能。事实上，大多数的此类工具都是运行在专用计算机内部的应用程序。

大多数运行局域网的家庭和组织，往往只会通过唯一的一点去连接到互联网上（然而，一些企业可能会拥有双重连接，如果他们可以为额外的可靠性的成本提供充足的合法理由）。这种单点连接是就各式各样的问题而进行通信消息的检查的一种非常理想的方案。提供这类功能的设施称为**防火墙**（firewall）。通常情况下，此类功能被嵌入与互联网进行连接的路由器上，因为不管以何种方式，路由器一直在查看着数据包。一个典型的防火墙配置如图 16-10 所示。一般而言，防火墙可以完成如下的一系列功能。首先，它可以通过查看相关连接请求所使用的端口号（port number）而阻止某些类型的所有连接。其次，它也可以检查数据包的内部，以拒绝接受某些类型的通信流——比如，ping 命令。再次，有些防火墙也可以配置成禁止那些来自于被认为是不安全或者令人讨厌的网际协议地址（即 IP 地址）的信息流。另外，防火墙还可以提供已知的与特定攻击相关联的数据模式，即所谓的**签名**（signature）。同时，它们还可以包含一个**通信监视器**（traffic monitor），用来监测那些与正常通信流模式存在显著差异的通信流模式。这种监测也被称为**异常检测**（anomaly detection）。采用签名和异常检测的网络保护系统通常被称为**入侵检测系统**（Intrusion Detection System, IDS）和**入侵防御系统**（Intrusion Prevention System, IPS）。图中的防火墙还显示了一个**隔离区**（DeMilitarized Zone, DMZ）。在这个案例中，此军事术语是指一个单独的网络，该网络既可以从外网进行访问，也可以从内网进行访问。换句话说，这便允许本地工作人员维护放置在该网络中的服务器的内容，同时也可以让那些服务器通过互联网方式进行访问。

378

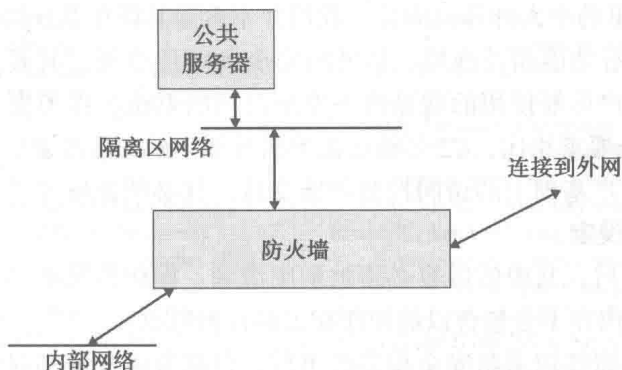


图 16-10 一个现代的防火墙

379

在网络连接点的防火墙存在一个问题，那就是并非所有网络中的机器都需要相同类型的连接。举例来说，一所拥有大型 UNIX 服务器的大学也许可能会希望防火墙允许建立 Telnet 会话，但是对于个人计算机来讲，可能就不需要这类 Telnet 会话。因此，在一台个人计算机中提供防火墙功能来禁止此类会话是十分常见的情形。非常有趣的是，个人计算机上的防火墙往往被设置成不仅阻止传入型连接，也会阻止许多传出型连接，因为连接到流氓主机来报告被盗取的信息是许多病毒惯用的一种技术。本地防火墙一般是一个应用程序，而不是操作系统的组成部分。但是，为了检查正在请求的连接以及传入和传出的通信流，本地防火墙需要能够在协议栈（protocol stack）中插入它们的功能。这对于构造协议栈的应用程序来讲，是一个不寻常的要求，但是，操作系统设计人员已经认识到其必要性，并为这一类特殊的应用程序提供了相应功能支持。正因如此，相关程序不仅能够监测消息中的文本模式，而且还可以监视其他的系统活动以检测具有病毒特征的行为，例如试图访问通常情况下应用程序不会访问的某些特定系统文件。

16.5 安全管理

为确保连接到网络的系统的安全性，系统管理员必须完成一些相关的事务。我们已经讨论了定期备份系统、防火墙以及通信流监测的需求。除此之外，应当把一些操作记录下来，并且要定期对相关日志进行审查。首先需要记录的是失败的登录尝试。有几次失败的登录情形是很正常的——每隔一定时间密码会过期，用户偶尔会敲错按键或者使用了其他系统的密码，等等。但是，失败登录数量的任何陡然上升，应当被作为是有人尝试侵入系统的可能迹象的警报的理由。其他的失败情形，比如在某网站服务器上无法找到所请求的网页，也应当被记录下来和进行分析。此类错误有可能只是表明错误的连接，但仔细检查也可能会发现是有人正在尝试攻击服务器。

与其他的系统相比，有些系统会拥有许许多多的更高的安全要求。除了阻挡病毒和黑客之外，一台家用的个人计算机可能并不需要多少安全特性。但一家银行的系统则需要更高的安全级别，因为相关系统会涉及金钱。进一步说，一家承担病人护理的医院的系统可能需要比此更高的安全性，因为系统相关问题可能甚至就会决定一个病人的生死。而一个军事系统则可能需要甚至还要更高的安全性，因为一次失败就可能会让数百万的生命遭受危险。鉴于此，美国政府下辖的一家机构，即国家计算机安全中心（National Computer Security Center, NCSC），发布了 4 种主要安全级别（另带有一些次级变种）的定义。伴随安全级别的逐级提升，相应的操作系统需要提供更多其他的功能，其中许多是在系统活动审计日志方面。不用说，对于家里的个人计算机而言，我们并不希望其操作系统因配备军事系统上满足相应安全所必需的所有功能而被拖垮。最低的安全级别是 D 级，其具有最少的安全性。办公室或家里的任何用户可能使用的就是这一安全级别的系统。作为安全级别递增的一个例子，同 C1 等级的安全要求相比，C2 等级还需要满足如下的一些要求：

- 建立在每个用户基础上的访问控制功能支持，其必须能够支持任何选定的用户群子集的访问权限设定。
- 存储空间使用后，其中的内容必须被彻底清除。操作系统必须确保分配给一个进程的磁盘空间和内存不会包含以前操作相关的任何数据。
- 操作系统必须能够记录与安全相关的事件，包括身份验证和对象访问。审计日志必须受到保护和可以防止篡改，同时必须记录日期、时间、用户、对象和事件。

现在, 大多数的商用操作系统很容易就能够达到 C2 级别。而运行在更高级别上的操作系统一般是针对特定目的而专门设计的。对于政府使用场合来讲, 操作系统的安全级别必须通过独立的第三方审计机构来进行认证。此外, 相应认证只会适用于操作系统的特定版本和特定的硬件配置, 因此, 开发商所做的普通的认证往往不会在这些更高安全级别上进行。

380

16.6 小结

现在, 鉴于计算机系统可以被许多用户所访问, 而且其也更加频繁地连接到互联网上, 所以, 无论是系统、其中的文件、其上面运行的进程, 还是用户之间的通信和进程之间的通信, 都应当受到保护和防止偶然的或故意的伤害, 这是非常必要的。在这一章中, 我们围绕操作系统讨论了保护与安全的若干方面。首先, 我们概要介绍了系统安全所涉及的一系列问题。我们就一些我们认为这是由于连接到互联网上而引发的安全问题进行了分类, 同时阐述了操作系统需要如何处理它们, 包括操作系统之外的相关机制。然后, 我们转去讨论操作系统为用户提供的保护服务, 主要是在文件隐私方面的保护。我们描述了这些服务的一般设计。同时, 操作系统还必须为进程提供相应的保护服务, 为此, 在运行进程与操作系统之间架起了重要的屏障。我们还阐明了提供给作为通信双方的进程的一些服务。在第 16.4 节, 我们讨论了网络安全, 大部分是针对互联网的, 具体包括加密、认证协议、消息摘要以及操作系统之外的与网络安全相关的一些话题。此外, 我们还介绍了网络及操作系统安全方面的管理问题。

在下一章, 我们将讨论一些特殊的事项, 当使用操作系统来创建跨越多个系统的分布式系统时, 我们必须就此予以充分地考虑和加以有效地处理。

参考文献

- Akl, S. G., "Digital Signatures: A Tutorial Survey," *Computer*, Vol. 16, No. 2, February 1983, pp. 15-24.
- Denning, D. E., "Protecting Public Keys and Signature Keys," *IEEE Computer*, Vol. 16, No. 2, February 1983, pp. 27-35.
- Denning, D. E., "Digital Signatures with RSA and Other Public-Key Cryptosystems," *Communications of the ACM*, Vol. 27, No. 4, April 1984, pp. 388-392.
- Dennis, J. B., and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM*, Vol. 9, No. 3, March 1966, pp. 143-155.
- Farrow, R., "Security Issues and Strategies for Users," *UNIX World*, April 1986, pp. 65-71.
- Farrow, R., "Security for Superusers, or How to Break the UNIX System," *UNIX World*, May 1986, pp. 65-70.
- Filipski, A., and J. Hanko, "Making Unix Secure," *Byte*, April 1986, pp. 113-128.
- Grampp, F. T., and R. H. Morris, "UNIX Operating System Security," *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, October 1984, pp. 1649-1672.
- Hecht, M. S., A. Johri, R. Aditham, and T. J. Wei, "Experience Adding C2 Security Features to UNIX," *USENIX Conference Proceedings*, San Francisco, June 20-24, 1988, pp. 133-146.
- Kramer, S. M., "Retaining SUID Programs in a Secure UNIX," *USENIX Conference Proceedings*, San Francisco, June 20-24, 1988, pp. 107-118.
- Lamport, L., "Password Authentication with Insecure Communication," *Communications of the ACM*, Vol. 24, No. 11, November 1981, pp. 770-772.
- Lehmann, F., "Computer Break-Ins," *Communications of the ACM*, Vol. 30, No. 7, July 1987, pp. 584-585.
- National Bureau of Standards, "Data Encryption Standard DES," NTIS NBS-FIPS PUB 46, January 1977.
- Needham, R. M., and M. D. Schoeder, "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM*, Vol. 21, No. 12, 1978, pp. 993-999.
- Organick, E. I., *The Multics System: An Examination of Its Structure*. Cambridge, MA: MIT Press, 1972.
- Reid, B., "Reflections on Some Recent Widespread Computer Break-Ins," *Communications of the ACM*, Vol. 30, No. 2, February 1987, pp. 103-105.
- Rivest, R., and A. Shamir, "How to Expose an Eavesdropper," *Communications of the ACM*, Vol. 27, No. 4, April 1984, pp. 393-394.
- Rivest, R., A. Shamir, and L. Adleman, "On Digital Signatures and Public Key Cryptosystems," *Communications of the ACM*, Vol. 21, No. 2, February 1978, pp. 120-126.

381

- Rushby, J. M., "Design and Verification of Secure Systems," *Proceedings of the 8th Symposium on Operating Systems Principles*, Vol. 15, No. 5, December 1981, pp. 12-21.
- Rushby, J., and B. Randell, "A Distributed Secure System," *Computer*, Vol. 16, No. 7, July 1983, pp. 55-67.
- Schell, R. R., "A Security Kernel for a Multiprocessor Microcomputer," *Computer*, Vol. 16, No. 7, July 1983, pp. 47-53.
- Silverman, J. M., "Reflections on the Verification of the Security of an Operating System Kernel," *Proceedings of the 9th Symposium on Operating Systems Principles*, ACM, Vol. 17, No. 5, October 1983, pp. 143-154.
- Simmons, G. J., "Symmetric and Asymmetric Encryption," *ACM Computing Surveys*, Vol. 11, No. 4, December 1979, pp. 305-330.
- Spafford, E. H., "The Internet Worm Program: An Analysis," *Purdue Technical Report CSD-TR-823*, November 28, 1988.
- Wood, P., and S. Kochan, *UNIX System Security*. Carmel, IN: Hayden Book Co., 1985.

网上资源

- <http://ciac.llnl.gov/ciac/index.html> (美国能源部, 首席信息官员办公室——计算机事件咨询能力)
- <http://www.cert.org> (卡内基梅隆大学计算机应急响应团队)
- <http://freshmeat.net> (网上关于UNIX和跨平台软件的最大索引)
- <http://www.ietf.org> (互联网工程任务组, 包括所有的互联网相关信息资料)
- <http://www.linuxsecurity.com>
- <http://www.netfilter.org> (Linux系统数据包过滤框架)
- <http://www.networkcomputing.com> (网络连接杂志)
- 在线版)
- <http://packetstormsecurity.org> (致力于保护网络安全的全非盈利性安全专业组织)
- <http://www.redbooks.ibm.com> (IBM出版物档案)
- <http://www.tasklist.org> (用于列出Windows系统上运行的所有进程的软件)
- <http://tldp.org> (Linux文档项目 (Linux Documentation Project, LDP))
- <http://www.usenix.org/publications/> (USENIX, 高级计算系统联盟)
- <http://en.wikipedia.org/wiki/Identd> (提供认证服务的守护程序)
- <http://www.windowsecurity.com>

习题

- 16.1 哪一个问题更为严重, 黑客还是内部人士? 请论证你的答案。
- 16.2 恶意软件区别于病毒程序的特征是什么?
- 16.3 恶意软件区别于特洛伊木马程序的特征是什么?
- 16.4 恶意软件区别于蠕虫程序的特征是什么?
- 16.5 简要说明缓冲区溢出。
- 16.6 沙箱模型的目的是什么?
- 16.7 身份认证利用了一些特殊的机制来验证实体的身份。我们在大多数时候所关注的是验证用户的身份。下面哪一项不是我们说过的可以用来验证用户身份的?
 - a. 你所持有的
 - b. 你所看到的
 - c. 你所知道的
 - d. 你所特有的
 - e. 上述所有选项均可用来验证用户身份
- 16.8 一旦用户(或其他实体)的身份得到验证, 用户所允许的操作必须被授权。我们讨论了通常用于支持授权的两种不同的机制。第一种是访问控制表。简要描述访问控制表是什么?
- 16.9 什么是权限表?
- 16.10 数据的备份副本应该存放在什么地方?
- 16.11 短语“蜜力攻击”是什么意思?
- 16.12 在对称密钥加密系统中, 使用了多少个密钥?
- 16.13 在非对称密钥加密系统中, 把两个密钥都保密好和不让外界知道是非常关键的。这是否正确?
- 16.14 如果我们不特别关注机密性, 但我们希望确保发送的消息不会被任何一方更改, 我们应使用哪

种机制？

- 16.15 认证机构用其自己的私钥对用户的公钥进行了签名。浏览器是如何使用相关机制来验证用户的公钥的？
- 16.16 就安全超文本传输协议 (HTTPS) 连接而言，网站层面使用了什么样的安全协议？
- 16.17 用来保护网络免受外部攻击的常见的机制是什么？

383
384

分布式操作系统

分布式系统正日益普遍和流行开来。我们讨论操作系统是因为其处于我们的应用程序和硬件之间。当我们开发非正式的应用程序的时候，我们并没有必要太多地担心操作系统。但是，当我们开发高性能应用程序时，我们就需要更好地了解操作系统的内部机制，以便我们能够与操作系统很好地协作，而不是与其发生对抗。于是，分布式处理应运而生。正如我们即将看到的那样，当我们要设计和开发支持大量用户的系统的时候，我们往往必须得开发分布式系统——拥有多个部分而且分别运行在不同机器上的系统。当然，我们也可能是出于性能或扩展以外的原因而去构建分布式应用程序，并且在这种情况下，我们可能仍然不需要了解太多的与分布式系统有关的操作系统的细节。但是，如果我们的系统是高性能的或大容量的应用程序，我们或许能够从了解底层服务的工作机理中不断地获得益处，那样我们就可以更好地利用相关机制，并且不会做出某些事情使得它们毫无目的地去完成额外的工作。

在本章开始的引言部分，我们主要讨论了为什么开发分布式系统的一系列原因，同时还介绍了分布透明性的概念及其重要性，最后引入了中间件的概念，就其演变过程及成因进行了解释。接下来，我们阐明了分布式系统所见到的若干不同的模型，其中既包括客户端服务器模型，也包括一些更为复杂的模型。第 17.3 节则回顾了进程和线程的相关话题，并讨论了如何在客户端和服务端中更好地使用线程从而改善和提高分布式系统性能的方法和技术。当分布式系统中的进程之间通信时，它们需要引用其他实体，为此在第 17.4 节中，我们讨论了命名的概念和命名空间。在第 17.5 节中，我们给出了分布式系统的一些不同的范例，包括远程过程调用、分布式对象和分布式文档。考虑到分布式系统所存在的一些特殊的与同步相关的问题，而且正是由于这些特殊问题使它们不同于单一整体系统，所以我们在第 17.6 节讨论了同步。其后在第 17.7 节中，我们讨论了容错以及分布式系统所具有的关于系统中一个部件发生故障而其他部分继续运行的特殊问题。最后，我们在第 17.8 节对这一章进行了归纳总结。

[385]

17.1 引言

我们可能需要开发分布式系统的原因有很多。我们曾经在第 9 章中就协作式进程展开了较为详细的讨论，因此我们在这里只是再次简明扼要地概括如下：

- 性能 (performance)。在多台机器上运行的系统拥有更多的处理器时间和其他资源可以用来解决相关问题。一些进程仅仅为了一次处理运行（譬如，模拟天气系统）就需要耗费很大的功率。
- 伸缩性 (scaling，或称为规模扩展)。多个系统意味着在给定的时间内可以处理更多的事务。
- 组件购买。很多时候，购买系统组件比在内部开发要便宜许多。有时，系统组件是以这样的方式开发的，也就是说，其基本上只能作为一个独立的进程加以使用，并且可能实际上需要一个单独的系统来运行。

- 第三方服务 (third-party service)。有时候应用程序组件需要访问特殊的数据库且它们自身并未被销售, 因此有关组件只能作为在线服务 (例如信用卡验证) 方式提供。
- 多个系统的组件。通常, 一个组件会在一个系统中开发, 但后来也需要作为其他相关系统中的一个组件。在这种情况下, 最好在一台专用的机器上隔离运行该组件。
- 可靠性 (reliability)。当系统仅仅拥有某组件的单个实例时, 该组件一旦发生故障便可能导致整个系统停止运行。而让每个组件拥有多个实例则会允许大型系统持续保持运行, 尽管这可能会带来性能的下降。
- 信息的物理位置。如果系统支持多个物理设施, 则可能希望系统各部分与相关设施是并置排列的。举例来说, 对于支持多个仓库的仓库库存系统而言, 其中大量的事务通常应用于本地数据库, 但是, 那些牵涉另一个仓库的少部分事务则需要建立相应的连接。
- 使应用程序的执行成为可能。一些应用程序需要非常多的资源, 如果没有高度分布的系统, 它们实际上是无法执行和完成的。外星文明探寻 (Search for Extra-Terrestrial Intelligence, SETI) 项目接受射电望远镜的大量信号, 并搜索可能标明智能起源的相关模式。项目组把射电望远镜信号划分成较小的数据集, 并将它们分发给那些通过屏幕保护程序对这些数据进行处理志愿者们。如果不是这样, 项目组实际上是没有办法处理这些数据的。

386

从理念上来说, 我们希望分布式系统可以实现若干方面的目标。首先, 分布式系统应当把用户和资源连接到一起。(注意, 在这里, “用户” 可以是另一个进程。) 其次, 系统应当表现出分布透明性 (distribution transparency) 的特征。理想情况下, 用户应当不会辨别出系统是分布式的。同时, 用户有可能从若干不同的方面注意到透明性的不足 (lack of transparency, 或称为透明性的缺乏)。具体包括如下几方面的透明性:

- 异构性 (heterogeneity)。不同的系统部件可能运行在不同的硬件系统上, 甚至无论硬件系统还是操作系统均各不相同的系统上。
- 访问。数据表示和访问的差异性 (浮点数据格式因机器而异)。
- 位置 (location)。资源所在的位置 (网页可能在任何地方)。
- 迁移 (migration)。资源是否可以移动 (脚本由服务器发送到你的浏览器)。
- 重定位 (relocation)。资源在使用时是否发生移动 (你的手机)。
- 复制 (replication)。资源被复制 (谷歌数据服务器)。
- 并发 (concurrency)。资源可以由许多用户共享 (网站)。
- 持久性 (persistence)。资源是保持在磁盘上还是在内存中。
- 故障 (failure)。资源在使用时是否发生故障 (互联网绕过故障链路选择路由)。

分布式系统的一个关键方面是它们严重依赖于相关开放标准 (open standard) 来实现所期望的透明性的绝大部分。计算机科学产业界中存在有许多标准, 有些标准是专有的, 而有些标准是开放的。专有标准通常在分布式系统中用处不大, 因为对于不同的供应商来说, 很难去测试相应组件的互操作性。故而, 由单个供应商为分布式系统所开发的许多操作系统机制往往最终被置为开启状态, 从而方便其他供应商可以测试其系统的互操作性。此类实例包括太阳微系统公司 (Sun Microsystems) 的网络文件系统 (Network File System, NFS) 和微软公司 (Microsoft) 的通用语言运行库 (Common Language Runtime, CLR)。

传统上, 大多数操作系统并没有支持分布式应用程序所需要的许多服务。于是, 这些服

务被开发放置在了所谓的中间件（middleware）上。如图 17-1 所示，从功能角度而言，中间件模块位于操作系统网络服务和应用程序之间。因此，操作系统和网络模块将为中间件提供服务，但在其他方面却忽略了中间件和应用程序之间的所有区别。网络服务可以彼此完全独立，并通过开放网络标准来进行通信。中间件模块也通过开放标准来进行通信，但是根据定义，它们相互协作以提供跨越系统边界的服务。

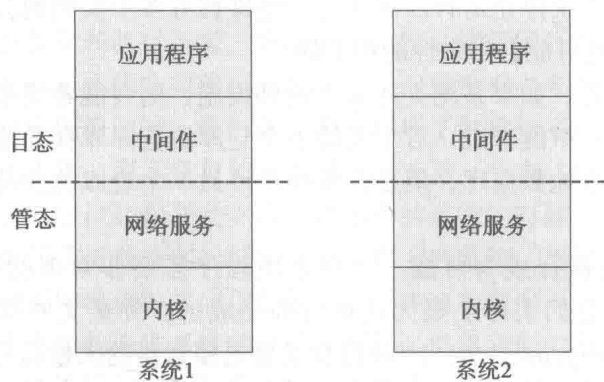


图 17-1 中间件服务层

17.2 分布式应用模型

显然，由运行在不同机器上的进程所组成的系统需要通信，并且已经有若干模型被开发出来，以用于描述这些组件之间的相互作用。在此，我们将主要介绍以下模型：客户端 - 服务器模型、三层模型、多层模型、水平分布和垂直分布。

17.2.1 客户端 - 服务器模型

客户端 - 服务器（client-server，或称为客户机 - 服务器）模型如图 17-2 所示。该模型可以说是众所周知，几乎不需要做任何解释。客户端系统需要访问某种定义明确的服务，故而与提供相应服务的服务器进行联系。我们在设计客户端 - 服务器模型时可能需要回答的主要问题是，对于应用程序的功能而言，应当在客户端部署多少，又应当在服务器上部署多少。一种极端方案是，有关应用程序运行在中央系统上，而客户端几乎就只是终端而已。这种模型有时被称为瘦客户端（thin client）。另外的一种方案则是，有关应用程序主要运行在客户端工作站上，而服务器仅提供非常有限的服务（例如数据库服务，即用来保存应用程序所使用的信息）。也有许许多多介于这两种方案之间的混合模型可以选用。我们将在下一节就此展开更为详细的阐述，相关原理与这里的模型基本一致，只不过是更多层级上进行运作而已。

17.2.2 三层模型

在使用客户端 - 服务器模型若干年之后，人们逐渐清晰地认识到，在大多数系统中容易辨认的功能实际上主要包括三个方面：用户界面（user interface）、应用逻辑（application logic，有时也称为业务规则，即 business rule）以及数据库存储（data-base storage）。这种体系结构模型（称为三层模型，

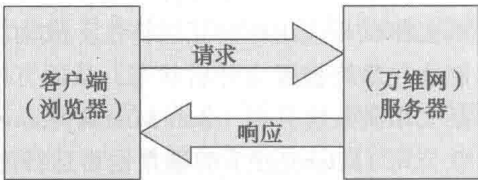


图 17-2 客户端 - 服务器体系结构

即 three-layer model) 如图 17-3 所示。之所以形成了这种特别的划分方案,其原因可能在于数据库系统自身的演化发展以及单独为每个应用程序构建此类设施很明显是不太合算的考量。

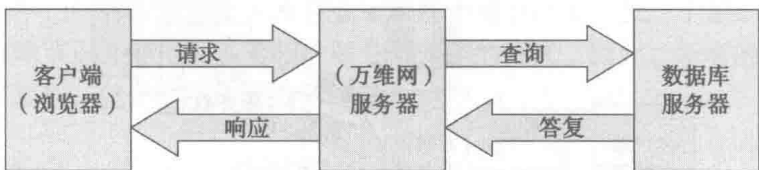


图 17-3 三层模型

与客户端-服务器模型一样,我们也可以对三层架构设计模型实施许多变化。用户界面可以做得非常简单,或许只相当于 UNIX 环境中的 X 终端 (X-Terminal)。对于其他的情景,个人计算机上的网络浏览器常常可以为应用程序提供一种呈现图形化用户界面的简单方式。在这种情况下,我们可以从万维网服务器发送一个页面,其中包含该应用程序的一张表单 (form) 供用户填写。有关用户填写对应表单,然后点击表单上的按钮 (button)。这一点点击操作将会导致浏览器把用户输入表单中的相关数据提交给服务器。于是,服务器检查数据,进而如果一切正常,服务器将会处理相关请求并返回对应的结果。但是,我们可以通过把某些处理迁移到客户端来,从而改善和提高有关设计的性能。进一步说,当用户把记录某种业务事件的数据输入表单中时,如果输入数据包含了某些可以检测出来的错误,那么越早捕捉到这样的错误并让用户加以及时的修正,则越好。例如,如果我们期望某字段包含日期信息,而用户输入了一些字母信息 (且并非月份名称),那么有关系统应当予以拒绝。如果我们一直等到用户提交表单并将其发送到服务器,然后我们再向用户发回错误消息,那么我们就将反馈与输入隔开了好几个步骤,从而使反馈的效用几近丧失。同时,这也中断和扰乱了用户的思维过程,因为用户已经在思想上从这个事务离开了,或者认为它已经完成了。换句话说,如果当用户把焦点从日期域移走的那一时刻就能够触发应用程序对数据格式的检查,则会更为合理、效果也会更好。总而言之,有相当多的设计工作常常是要决定在客户端可以做什么样的检查,以及在服务器端应该完成什么样的任务。

还有一些其他的功能也可以迁移到客户端来。例如,由于通信成本可能很高或者网络连接不太可靠,所以如果允许客户端在离线情境下完成相当数量的数据收集并且稍后当服务器或相关连接再次可用的时候将对应事务数据再提交给有关服务器,那将可能是非常有益的。

第三层,即数据存储,往往由封装的数据库管理系统予以提供。通常情况下,这些系统只是提供了一种更高级别的文件系统,支持非常可靠的数据存储以及规范化表格中的数据检索。对于另外一些案例而言,数据库系统还通过运行存储在数据库中的过程而被用来执行一部分系统逻辑,通过验证参照完整性和整理报告数据等来改善数据有效性。

17.2.3 N 层应用程序

三层模型常常会扩展到 N 层,从而形成多层模型。这有时被称为垂直分布 (vertical distribution),一般在应用程序可以被方便地划分成若干部分的情况下就会采用这种结构。一个具体的例子是谷歌搜索引擎的架构,如图 17-4 所示,谷歌系统被分解成了客户端 (浏览器)、(万维网) 服务器、拼写检查器、广告服务器、索引服务器以及文档服务器等若干部分。虽然关于该体系结构,尚未找到相关的详细描述,但已经公布的内容足以说明这一点。

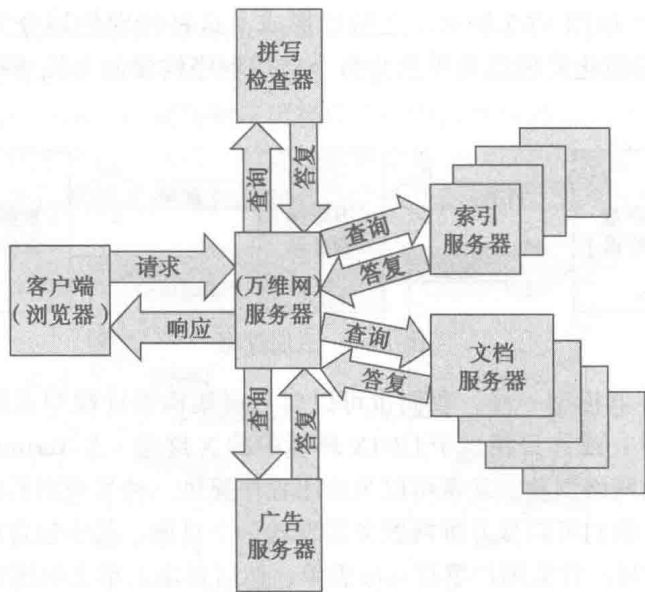


图 17-4 谷歌系统体系结构

其中，有一个前端进程用来接收相关请求并将该请求解析成为一个个孤立的单词。另一服务器则可用于相应的拼写校正。接下来，前端服务器就可以接收到这些单词并把它们传递给其他服务器，而这些服务器分别负责包含给定单词的网页索引的数据库。进一步说，这些服务器将会把一组指针传递到另一台服务器上，而后者则会合并相关指针集合，从而创建指向满足完整的搜索条件的页面集合的一组指针。通常情况下，此类搜索会包含查询中的所有单词，但是也有可能是其他形式的查询。还有一个服务器是被用来实现从数据库中提取广告信息的查询的，也就是选取与搜索字词相关的广告或者选取与该用户以往所做的其他搜索相关的广告。有关页面的排名次序决定了它们与用户查询之间的可能的相关性，并且相关指针用于从其他服务器获取被缓存的页面，从而使得被引用页面的短的片段可以被合并到即将被返回到客户端浏览器的网页中。因此，我们在这个应用程序中至少看到了5个不同的层级。虽然它们可能并不是严格意义上的垂直分层，但它们是相互作用的组件，而且这些组件是服务于许多客户端的独立的服务器。

17.2.4 水平分布

我们还会看到，另一种范式也被用于以两种不同方式构建起来的谷歌系统中，即水平分布（horizontal distribution）。我们曾经提到，有关数据库分布在一组服务器上，每台服务器只负责包含指定单词（或单词集合）的页面。这种布局是一种分布式数据库，其中一部分信息包含在了一台服务器上，而一部分信息则包含在了另一台服务器上。此外，谷歌所使用的服务器是比较廉价的个人计算机，而不是高性能的计算机。关于计算机的确切数量并不为人所知，且有关估计各不相同。不过，据一家研究机构估计认为，谷歌系统在2007年中期拥有100万台服务器，并且以每季度多于10万台的速度不断增加。谷歌认为服务器会发生故障，为此，处理包含特定字词的文档的服务器实际上是由若干台服务器所组成——在给定的一个谷歌网络节点上至少有三台服务器。进一步说，字词数据库往往会被复制到至少三个地理上分布的节点中，以便限制由于包含节点的中心所发生的灾难而导致的故障。这被称为复制型数据库（replicated database）。综上所述，谷歌数据库既是分布式的，又是复制型的。

然而，我们之前提到的其他各类服务器也存在这样的情况，即它们也并非分别由唯一的一台服务器所组成。在这个世界上，可能没有任何服务器能够跟得上谷歌每小时所接收到的搜索请求的数量。为此，整个网络被设计成了有关请求在一组基本相同的搜索引擎之间进行分发的方式。同样，广告服务器和拼写检查服务器也拥有许多实例。于是，整个系统被设计成了绕过任何故障节点并使用另一台相应服务器实例的方式。因此，所有的各种各样的服务器功能都得到了复制，就像数据库服务器一样。

17.3 抽象概念：进程、线程和机器

进程是操作系统用来对处理器进行虚拟化的一种抽象概念，其目的在于使一道正在运行的程序无须了解或意识到其并没有真正地控制处理器。为了让系统在一道应用程序上完成更多的工作，我们可以让一个进程创建其他的进程，新创建的这些进程也将运行，从而将会在处理器上获得额外的处理器执行周期。然而，从一个进程切换到另一个进程，需要在单处理器系统（uniprocessor system）上执行上下文切换（context switch），并且上下文切换会导致系统性能的严重下降。所有的高速缓存或者必须被刷新，对于用来高速缓存页表项的快表尤其如此，或者将不会找到任何的高速缓存项，直到新进程运行足够长时间后对高速缓存完成了关于新进程的重新加载。因为不能命中快表，所以将会导致系统速度的下降，这种情况往往一直到对应快表被重新加载从而映射到正在启动的进程的完整的工作集（working set）时才会得到解决。

鉴于此，便开发和形成了线程的概念。进一步说，线程源自于人们关于进程控制块中所持有的状态信息实际上是由两部分组成的认识。其中一部分代表有关进程当前所持有的诸多资源，另一部分则保存了关于处理器当前执行点（对于实际上没有运行的任何进程而言）的真正的处理器状态。后一部分的存储可以被复制，并且第二份这样的状态信息可以用来跟踪处理器关于同一进程的另一不同的执行点。为此，一道程序在对应进程若干部分也在运行的情况下，便可能实际上要求操作系统允许该进程的其他部分继续运行，只要这些部分可以正常通信和同步相应操作就行。于是，这便允许一道程序拥有多个并行的执行点，并且不会引发上下文切换的开销。

17.3.1 线程

线程可以采用若干有益的方式而被运用在分布式系统中。在客户端系统中，线程可被用来支持处理操作与异步通信的相互重叠。一个主要的例子是运行 1.0 版超文本传输协议的网络浏览器。在超文本传输协议的早期版本中，浏览器首先会获取到文档的基页（base page）。伴随浏览器对文档中的嵌入式元素的扫描，其接下来不得不单独连接到有关服务器，逐个地去提取每项其他的元素。故而，使用对应协议的浏览器可以启动单独的线程来检索每项元素，而不是逐个直接地去获取它们。这便极大地提高了有关进程的速度。类似地，正在执行一长串远程过程调用（Remote Procedure Call, RPC）的客户端可以把每个调用放在一个独立的线程中进行，其前提条件是在一个调用中不需要另一个调用的结果。

服务器端也可以充分地利用线程。其中，主要的用途是在单独的线程中处理每项传入的请求。一开始，系统启动一个主线程（primary thread）或调度线程（dispatcher thread），用来侦听所传入的请求。当一项请求到来时，主线程将会启动一个工作线程（worker thread）来处理相应的请求。这种设计同时还具有程序简单性的优点。假设我们采用了内核级线程

(kernel-level thread)，如果有关工作线程执行了一个阻塞内核的调用，或许是要读取磁盘吧，那么仅仅是对应的线程发生阻塞，而服务器的其余部分可以继续工作。每个线程都通过一系列（通常）简单的步骤来处理相关请求、返回对应结果并且终止。如图 17-5 所示。

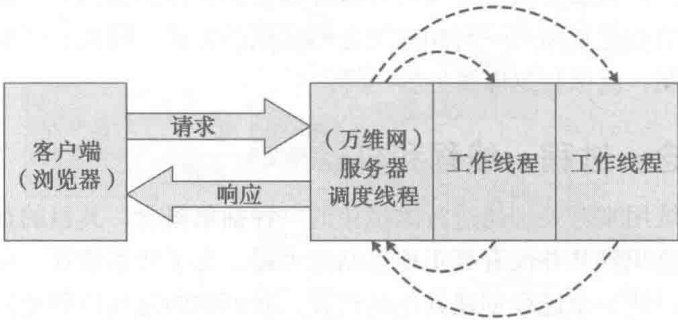


图 17-5 多线程服务器

17.3.2 虚拟机

虚拟机是另一层级的抽象——虚拟化整台机器，而非仅仅虚拟化处理器。进一步说，有两种不同类型的虚拟机（Virtual Machine，VM）。对于首字母缩略词 VM 而言，这可以说是一个令人可叹的重载，因为该缩略词也用于指代虚拟存储器（Virtual Memory，或称为虚拟内存）。当然，这两者之间的不同通常从上下文情景还是很容易区别开来的。

物理虚拟机

首先，我们来介绍一下物理虚拟机的概念。一开始，一个小的操作系统内核被加载，然后将会在该操作系统内核基础之上执行其他的操作系统。最初被加载的操作系统称为主机操作系统（host OS），而后来执行的其他操作系统则被称为客户机操作系统（guest OS）。这里的基本技巧是，当主机操作系统加载客户机操作系统时，将会使后者运行在用户模式。于是，每当客户机操作系统试图要执行任何通常要求管态模式的操作时，硬件将会触发一个中断并被主机操作系统所接收，然后主机操作系统将会执行相应操作，并且当完成有关操作时，将会把对应结果返回给客户机操作系统。如图 17-6 所示。关于在同一时间在同一台机器上运行多个操作系统能够派上用场的理由有好多。就分布式处理而言，其主要原因是把多台服务器合并到一个系统上。构建一台非常可靠的高性能的服务器，并将其放置在一个安全的位置是相当昂贵的。通常情况下，现在购买的一台服务器往往会比运行相应服务实际所需的功能要强大得多。采用虚拟机将可以支持多台服务器的整合。这便可以节省硬件上所花的开销，因为一台较大的服务器可以替代若干台较小的服务器，并且消耗的功率和冷气都将大大减少。如果各服务器运行在不同的操作系统平台上，将会特别有用。不过，即使各服务器运行在同一种操作系统上，虚拟机也可以运行任何客户机操作系统的多个副本。这看起来有点怪怪的，但是这确实有助于服务器功能的隔离，因为这样，一个客户系统的崩溃将不会影响到任何其他客户系统。

[392]

抽象虚拟机

抽象虚拟机是另一种类型的虚拟机，其被设计为运行某种中间语言（intermediate language）的机器的软件模拟。对应主要的例子是由太阳微系统公司（Sun Microsystems）开发的 Java 虚拟机（Java Virtual Machine，JVM）以及由微软

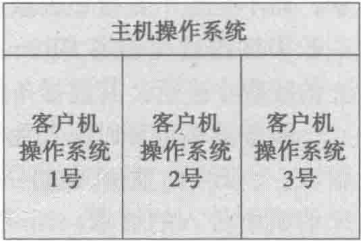


图 17-6 物理虚拟机

公司 (Microsoft) 开发的作为 .NET 系统的组成部分的通用语言运行库 (Common Language Runtime, CLR)。它们在分布式处理中得到了广泛的运用, 且主要是出于三方面的原因: 代码移动性 (code mobility)、代码可移植性 (code portability) 和安全性。就代码的移动性而言, 其允许将编译后的程序从服务器下载到客户端从而在本地运行。当从一台服务器下载 Java 小程序 (Java applet) 并在客户端浏览器中运行时, 便属于这种情况。在客户端浏览器中包含有 Java 虚拟机的实现, 因此 Java 程序可以从服务器复制过来并在客户端运行, 这可能是由于我们前面所提到的某种原因。对于运行在虚拟机中的代码, 其可移植性更强, 因为虚拟机本身就可以被移植到任何硬件和平台上。因为目标机器是虚拟的, 所以向软件供应商保证了广阔的市场。不过, 鉴于 Java 虚拟机可能正在客户端的浏览器中运行, 所以我们可能会有一些风险, 即下载的小程序可能会出现安全问题。因此, 默认情况下, 浏览器往往会严格限制有关小程序可以实施的行为——譬如禁止访问本地硬盘驱动器。一般来说, 客户端浏览器可以被配置为指示某些特定的站点是可信的——譬如客户端公司总部, 并且来自相应站点的代码将被允许完成某些事情, 而这些事情是不允许源自其他站点的代码来执行的。相对于在浏览器的虚拟机仿真平台上的执行而言, 另一种替代方案是, 有关程序可通过一种称为即时 (Just-In-Time, JIT) 编译的操作而被编译成目标机器的本机机器代码 (native machine code)。

[393]

17.4 命名

分布式应用程序需要在所牵涉的各种各样的进程之间进行通信。当进程通信时, 它们需要引用其他的实体, 譬如文件、套接字、记录、用户, 等等。对实体的引用可以采取多种形式。我们需要区分清楚名称、标识符和地址。名称代表实体。用户有名称。计算机系统也有名称——例如, webserv.example.com。名称可以被重用, 故而对于刚才所举的域的例子来说, 有可能会将当前称为 webserv 的一个系统替换为另一个系统, 从而调用了新的 webserv 系统。这对于任何类型的服务器尤其可能如此。我们可能期望访问的许多其他的实体也拥有名称。名称不一定是唯一的。因此, 我们创建了真正的标识符 (true identifier)。一般来说, 标识符是由权威机构发布的。标识符从不重复使用, 也永远不会被复制, 故而始终指向同一个实体。具体例子包括用户的社会保险号码或网卡中烧录的介质访问控制地址。最后, 地址代表实体的接入位置。具体例子包括电话号码、网际协议地址和套接字 (或端口), 末者常常用来寻址计算机系统内的一个特定的进程或线程。

引用在单一系统中的传递往往非常简单, 因为整个系统共用一套通用的引用框架。因此, 对于大多数的平台而言, 一个没有任何其他上下文的简单的文件名, 首先将会被假定在当前工作目录 (current working directory) 中。如果在当前工作目录中没有找到, 那么将会使用一系列备选的目录。它们通常在为系统 (或为给定用户) 所定义的某组全局的取值中加以指定。在微软的操作系统上, 这组取值被称为环境变量 (environment variable), 且其中之一被称为路径 (path)。对应路径是操作系统用来查找不带路径名的一个文件时将会搜索的一系列目录。这些备选目录共同组成了一个通用的引用框架, 且其中的文件名具有意义。系统上的所有进程往往会共享这一引用框架。

由于这一通用的引用框架, 所以实体引用在单一系统中的传递往往并不困难。然而, 分布式系统则要复杂许多。这是由分布式系统的异构特性所造成的问题之一——它们一般不会分享通用的引用框架。为了提供通用的引用框架, 业界已经建立了一些全局性的引用框

394

架，常常称为名称空间 (name space)。名称空间是有组织的信息集合，其中，名称可以被定位。主要的例子是域名系统 (Domain Name System, DNS)。域名系统是由互联网命名机构 (Internet Naming Authority, INA) 所定义的分层式名称空间。对于进程来说，通过传递域名系统名称进行查找并发现与相应名称所对应的网际协议地址是非常简单的。通常情况下，对应进程往往会拥有一个与网际协议地址一起使用的套接字号 (socket number)，并且这一连接到一起的数字确定了所寻址的系统中的特定的软件实体。

17.4.1 发现服务和 Jini

发现服务 (discovery service) 是分布式系统的一项重要功能，而 Jini (发音类似于 genie，即精灵) 则是一种用于动态创建分布式系统并完成发现服务的中间件设计。它是支持开发人员创建高度适应变化的无论硬件还是软件的网络服务的一种开放式规范。这种设计具体规定了客户端在网络上查找服务，然后使用相关服务完成任务的方法。进一步说，服务提供者向客户端发送相关 Java 对象，从而为客户端提供对相应服务的访问。这种交互可以使用任何中间件技术，因为客户端只会看到对象，并且所有的网络通信仅限于该对象以及其所访问的服务。

当一项服务加入启用 Jini 的网络时，该服务便通过发布实现已知服务的应用程序接口的对象的方式来公布自己。而客户端则通过查找支持其想要使用的应用程序接口的对象来查找有关服务。于是，当客户端发现对应服务的已发布对象时，便可以下载用来与对应服务进行通信的相关代码。

17.4.2 发现服务、X.500 以及轻量级目录访问协议

目录访问协议 (Directory Access Protocol, DAP) 是由国际电信联盟电信标准局 (International Telecommunication Union-Telecommunication Sector, ITU-T) 和国际标准化组织 (International Organization for Standardization, ISO) 所规定的用于与 X.500 目录服务一起使用的网络标准。其目的是被客户端计算机所使用，但是并不太成功，因为几乎没有用于个人计算机的开放系统互连 (Open System Interconnect, OSI) 协议簇的实现。目录访问协议的基本操作被包含在了 Novell 目录服务 (Novell Directory Service, NDS) 中，后来还被包含在了轻量级目录访问协议 (Lightweight Directory Access Protocol, LDAP) 中。

轻量级目录访问协议旨在成为访问 X.500 目录服务的轻量级替代方案，并且能够运行在传输控制协议 / 网际协议的上面。进一步说，其目的在于客户端可以通过从轻量级目录访问协议到目录访问协议的网关 (LDAP-to-DAP gateway) 来访问 X.500 服务。然而，基于轻量级目录访问协议的目录服务器迅速兴起。轻量级目录访问协议受到了企业界的普遍欢迎。它成为 Windows XP 的默认目录服务，并且也可方便用于现今大多数其他的操作系统。轻量级目录访问协议包含有非常强大的认证协议，故而对于分布式服务的访问是十分安全的。

17.4.3 对移动式网际协议的实体的定位

如果设备是移动的，那么它们在互联网上使用网际协议进行通信时便会产生一个特殊的问题。这一问题的出现是因为节点的网际协议地址的一部分指定了对应节点所连接的网络。如果该节点移动到另一不同的网络，那么相应的网际协议地址应该发生变化。但是，传输控制协议连接模型和大多数其他的协议并没有设计为允许在会话期间改变网际协议地址。因

此,跟踪移动式网际协议 (mobile IP) 的实体是十分困难的。移动式网际协议在无线环境中可谓是司空见惯,例如,当用户从家里到学校去工作时,有关用户将会带着他们的移动设备跨越多个网络。

395

针对移动式网际协议的协议簇由 3344 号互联网标准草案 (Request For Comment, RFC) 给出了具体规定。一般来说,要采用移动式网际协议的节点往往会拥有自己的一个称为主地址 (home address) 的网际协议地址。该节点将会在自己的主网络 (home network) 服务器上进行注册,对应服务器被称为移动式网际协议主代理 (IP home agent)。当该节点移动到另一个网络时,它将会被赋予一个在新的网络上的网际协议地址。这被称为转交地址 (care-of address)。然后,该节点将会搜索一个称为移动式网际协议外部代理 (IP foreign agent) 的服务器,并把自己的主代理的位置告诉该外部代理。于是,该外部代理便会连接到对应节点的主代理上,而相应的主代理将会把这一临时的新的网际协议地址存储到一个数据库中,并且将在当地用该网际协议地址来注册自身。一台需要与该移动节点进行通信的主机在一开始连接到了该节点的主地址上。故而,当有关数据包被主代理所接收后,将会由该主代理利用新的网际协议报头把相应的数据包转发到对应移动节点的转交地址,且原先的网际协议数据包被保留在了新的数据包中。而该节点中的移动式网际协议软件将会剥离掉外层的数据包报头,并把内层的数据包传给该节点中的应用软件。这一过程被称为网络隧道 (tunneling)。有关应用软件无须关心自己正运行在移动环境中 (也就是说,相关中间件提供了移动透明性)。

17.5 其他分布式模型

我们在前面讨论了客户端-服务器模型以及在该模型基础上的若干变种,而在分布式系统中,还有其他的一些模型也是十分有用的。

17.5.1 远程过程调用

现有的单机系统 (monolithic system) 常常需要修改成为分布式系统。一种分离现有进程的模型就是从现有的应用程序中移出某些子例程并让它们运行在一台单独的服务器上,称之为远程过程调用 (Remote Procedure Call, RPC)。这是一项很有用的技术,因为它融入了程序员业已熟悉的组件模型 (component model)。从原理上讲,有关思想非常简单——从一个运行系统中取出一个子例程并将之部署在一台服务器上。其间,用来替换被移出例程的新的例程被称为客户端桩 (client stub) 例程。进一步说,客户端桩例程知道相应真正的子例程位于其他某处地方,并将触发远程过程调用的中间件来查找和调用该子例程。相关处理模型如图 17-7 所示。但是,分布式系统可能存在的异构特性却使得有关过程颇为复杂。用来定义远程过程调用的 1831 号互联网标准草案假设有关系统是异构的,这便意味着传递给子例程的参数必须得转换为运行对应子例程的服务器的格式,并且在返回时必须得以相反的方式来转换对应返回值。这一过程被称为编组 (marshalling) 和解组 (unmarshalling)。同时,在服务器系统上应当设立另外的一个桩例程,而且,从其调用对应子例程的角度来讲,该服务器桩例程顶替了原始程序的作用。具体而言,服务器桩例程接收来自于客户端系统的消息,把有关参数解组成为服务器平台所要求的格式,然后调用对应子例程,接下来解组所返回的参数,最后将它们打包成消息和发给客户端桩例程。

鉴于客户端系统并不清楚服务器系统是运行在什么平台上,所以客户端桩例程将把有关参数

396

转换成由 1832 号互联网标准草案所定义的称为外部数据表示 (eXternal Data Representation, XDR) 的中间格式。此中间格式是与平台无关的, 允许我们以一种标准化的与平台无关的格式来表示任何数据。不过, 对于给定平台的远程过程调用实现则必须要定义从外部数据表示格式到本机平台格式的相关映射。

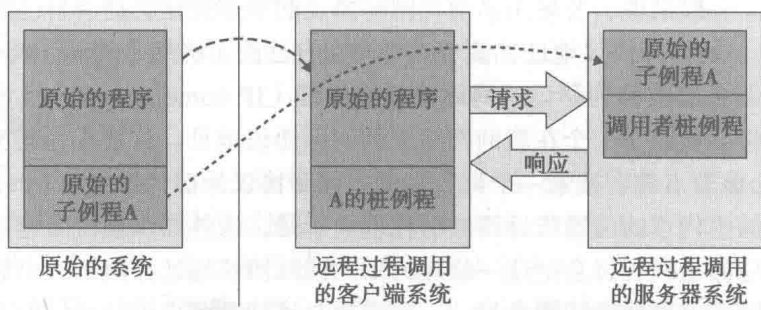


图 17-7 远程过程调用模型

当从程序中移除一个子例程的时候, 必须得用相应的客户端桩例程来加以替代。通常, 从一种所谓的接口描述语言 (interface description language, IDL) 入手来创建有关的桩例程。大多数的接口描述语言与 C 语言相类似。桩例程用于声明被移出的例程的参数类型。一旦采用接口描述语言对有关接口进行了描述, 那么接下来就应针对相应描述来运行一个特定于客户端平台和源语言的接口描述语言编译器。这将会产生两样东西, 具体包括一个即将被插入原始程序中用以描述所缺失的例程参数的头文件以及一个应当被编译的单独的源程序, 且后者将会成为客户端桩例程。相关过程如图 17-8 所示。该例程的目标格式将与原始应用程序进行链接从而生成修改后的应用程序。相应的接口描述语言是标准化的, 因此同样的接口描述语言文件可以被用来在服务器平台上生成服务器桩例程以及被该桩例程所调用的修改后的例程。

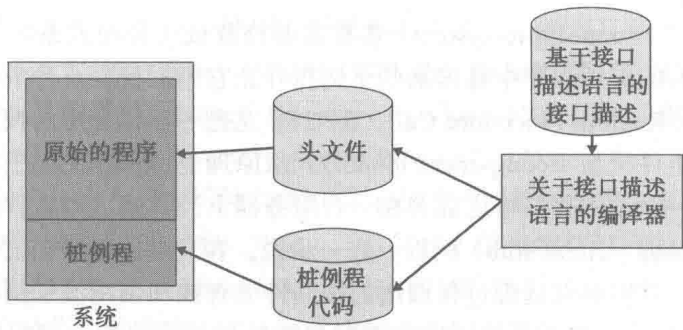


图 17-8 远程过程调用的桩例程的创建

如前所述, 远程过程调用由互联网标准草案进行了定义, 而且, 有关规范在具体的软件包中得到了实现。进一步说, 其中就包括由开放集团 (Open Group, 也就是对象管理组 (Object Management Group, OMG)) 作为开源项目所创建的称为分布式计算环境 (Distributed Computing Environment, DCE) 的一种颇为标准化的实现方案。毫无疑问, 各家系统制造商能够创造出他们自己的软件包实现方案, 但是, 采用这种开源代码所拥有的优势在于, 相关软件包是经过严格测试的。毕竟, 开放集团是由超过 800 家组织所构成的特设集团。

17.5.2 分布式对象

分布式对象模型是与远程过程调用非常类似的一种模型。有关技术非常相像，但是对象要比子例程更为复杂。同时，相关组件的命名也略有不同。在客户端系统上的桩程序被称为代理程序（proxy），而在服务器端的桩程序被称为程序框架（skeleton）。通常情况下，在服务器端还会发现一个称为对象适配器（object adapter）的额外的组件。其功能是就如何调用有关对象实施一些管理方面的限制，最常见的限制如串行化器（serializer），即限制有关对象每次只能调用一个，除非已经确认相应对象是线程安全的。

与远程过程调用一样，分布式对象也由相关规范进行了定义，并且在特定的软件包中予以实现。通用对象请求代理（Common Object Request Broker, CORBA）体系结构是关于分布式对象的主要标准之一，且该项标准也是由开放集团所定义的。在有关体系结构中，中间件层本身拥有一个特定的名称，即对象请求代理（Object Request Broker, ORB，按“orb”发音）。

17.5.3 分布式文档

分布式文档（distributed document）是在分布式处理中所使用的另外一种模型。毋庸置疑，万维网（World Wide Web, WWW）是关于这一模型的最为知名的实例。最初，万维网只是作为一种提供方便获取与原子能相关的研究论文的机制而被创建起来的。它包含了一种称为超链接（hyperlink）的引用其他文献的技术，尽管这并非一种全新的概念。在文本文档中也常常可能嵌入链接来引用图形源文件，即包含相应图形的原文献。当然，现在这一模型已经得到了极大的发展，并且比文档中的链接要复杂许多。事实上，我们通常会提到“网页”而不是文档。现今的想法是，一个网页最多只能包含几千个词，并且应当根据需要链接到其他的网页上。如今的网页不仅可以包含到诸如声音和电影等其他多媒体元素的链接，也可以包含关于表单的链接，从而支持用户输入信息以与在服务器上运行的应用程序来进行交互。更为重要的是，网页现在可以包含譬如脚本和小程序在内的可编程元素。为此，可能需要采用那些原本设计是为了创建网页的工具来开发非常复杂的应用程序。此类接口的最大优势在于客户端仅仅需要一个浏览器来访问应用程序的相关功能。从理论上来说，这便意味着应用程序网页不仅可以通过个人计算机来进行访问，而且还可以通过掌上电脑和手机来加以访问。不过，相关使用也存在一些困难，主要牵涉访问速度及屏幕尺寸问题。因为，现今大多数网站是基于较大的用户屏幕和快捷的连接速度等假设前提而设计的。

[398]

采用文档模型的另外一种不太知名的系统是 Lotus Notes。它是一种按照各种主题把相关便笺库加以曝光的高度复杂的应用程序。某些主题会通过电子邮件等方式自动推送给所有用户。而另外的一些主题内容则只有在相应用户请求时才会被访问。尽管并没有非常多的机构是 Lotus Notes 的用户，但有关机构却往往是拥有许多用户的大型组织，因此在采用文档模型的分布式系统的任何讨论中，该系统通常至少值得一提。

采用文档模型的其他系统还包括互联网电子邮件和网络新闻。它们各自以不同的方式进行运作，并利用相应的推送和拖拽协议及特定的途径向客户端分发信息。

17.5.4 分布式文件系统

文件是所有的程序设计人员以及大多数的用户都理解的一个概念，于是自然而然地，许多利用文件系统以某种方式连接机器来支持分发服务的系统被开发出来。我们曾经在第 7

章中提到过网络文件系统模型，其后，我们在第 13 章中对该模型进行了更为细致的分析研究，为此我们在这里不再重复相关讨论。网络文件系统支持远程机器上的目录出现在本地系统上，就像它们是本地的目录一样，从而提供位置透明性。网络文件系统由太阳微系统公司（Sun Microsystems）在 UNIX 系列操作系统上开发，故而目前也可以在 Mac 系列操作系统和 Linux 操作系统中加以使用。微软公司也为 NT 系列较高版本的操作系统提供了可选的网络文件系统客户端和服务器的支持。

微软公司还提供了一种类似的服务，冠以通用互联网文件系统（Common Internet File System, CIFS）和服务器消息块（Server Message Block, SMB）等各种名称，它们均类似于网络文件系统。在此基础上，还有公司基于逆向工程技术为非微软操作系统开发出了与此兼容的客户端和服务端，称之为 Samba。

网络文件系统和通用互联网文件系统均要求建立从客户端到服务器之间的不透明的连接，而其他的系统则是为了使得有关过程更加透明才开发出来的。例如，微软公司就此开发出了一一种所谓的分布式文件系统（Distributed File System, DFS），用来构建文件服务器和共享目录的分层式视图，从而使它们可以分别被赋予唯一的命名。于是，用户仅仅需要记住一个名称，而不再被由一大堆不同名称所组成的链接搞得晕头转向。分布式文件系统支持服务器复制，同时将会为一个客户端建立其到达最近可用的文件服务器的路由。分布式文件系统还可以安装在集群系统上，以获得更好的性能和可靠性。

另外也有一些其他的不太广泛使用的分布式文件系统，其中特别包括安德鲁文件系统（Andrew File System, AFS）和内容分发体系结构（Content Delivery Architecture, CODA），二者均由卡内基梅隆大学所开发。安德鲁文件系统是为每个客户端工作站提供一个同构的、位置透明的文件命名空间而设计的。内容分发体系结构作为一种更新的产品，则把重点放在故障恢复和断开连接操作（移动计算）方面。相关系统只有在 UNIX 及其衍生操作系统中才得到支持。

399 还有，谷歌搜索引擎的设计也严重依赖于分布式文件系统的体系结构，其在所有系统中都提供了三重冗余。遗憾的是，对于我们来说，有关设计方案是专利性质的，尚无法找到多少详细的相关信息可以使用。

17.6 同步

正如我们在第 9 章中所见到的那样，对于由诸多模块组成的系统来说，相关模块之间需要同步彼此的操作。而就分布式系统来讲，甚至需要加倍努力才能够实现和提供相应的同步操作。在此，我们将会讨论分布式时钟（须确保时钟同步）、同步、互斥、协调器选举和并发控制等若干种机制。

17.6.1 时钟

在许多分布式算法中，要求获悉有关事件的先后次序。譬如说，如果有两个人几乎在“同一时间”从银行账户执行了取款操作，那么我们往往希望在实施第二个人的取款操作之前就已经完成了第一个人的取款操作。遗憾的是，光速限制了从一个系统到另一个系统的传输时间。因此，即使两个事件的确是在同一时间发生的，也只能到一段时间之后才可能知道这一点。除此之外，这也使得几乎不可能真正确定两个系统上的时钟是同步的。幸运的是，通常我们并不真正关心两个事件发生的实际时间，我们只是关心事件发生的先后次序。

这便使得有关问题得到一定程度的简化。进一步说,我们真正需要的是**逻辑时钟**(logical clock)。逻辑时钟背后的想法是,存在某事件集,对于其中的事件而言,我们主要关注的是它们发生的先后顺序。为此,在每个系统中,每当有事件集中的某个事件发生的时候,我们就对某个计数器执行递增操作。我们把当时的计数器的值与该事件相关联,称为**时间戳**(timestamp)。这便形成了我们即将用来对事件进行排序的逻辑时钟。对于两个事件而言,如果一个事件的时间戳小于另一个事件的时间戳,那么我们就认为前一个事件是在另一个事件之前发生的。

在分布式系统中,我们还必须采取另外一项措施。也就是说,我们必须关注进程之间的消息。我们希望确保消息的发送事件是在相应消息的接收事件之前发生的。为此,我们需要把消息发送的时间戳与另一个进程进行关联,并且将该逻辑时钟值与对应消息一起发送出去。接下来,当该消息被接收到时,相应的接收系统将会检查伴随消息发来的逻辑时钟值。如果传入消息的逻辑时钟值大于接收进程的逻辑时钟值,那么接收进程应当把自己的逻辑时钟设置为传入消息中的逻辑时钟值加 1,从而表征相应消息的到达事件。否则,接收进程只是简单地对自己的逻辑时钟执行加 1 操作以表征消息的到达。这种机制称为**兰伯特时间戳**(Lamport timestamp)。

遗憾的是,有时候我们所需要的比这还要复杂。通常情况下,我们需要了解在其他系统上的哪些事件可能会对传入消息所描述的事件产生影响。为此,用来记录和跟踪分布式系统中的所有进程的时间戳的机制是为每个事件附着上一个**时间戳向量**(vector of timestamp)。该时间戳向量的索引是由分布式系统分配给每个进程的一个编号,并且该向量中第 i 项的取值就是我们所获悉的关于该进程的最新的**时间戳**。当发送消息时,不是将本地的事件计数器的取值作为时间戳进行发送,而是把该进程所获悉的整个时间戳向量与消息一起全部发送出去。当接收系统接收到相应消息时,对于传入消息时间戳向量元素取值大于其自身时间戳向量对应元素取值的情况,应当根据前者来修正和更新后者。

400

17.6.2 互斥

当两个进程协作时,它们经常需要同步对共享数据的访问,以避免相关更新操作的冲突问题,这部分同步称为**互斥**(mutual exclusion)。进一步说,在每个进程中,往往会包含一个称为**临界区**(critical region)的代码段,其中会牵涉相应共享信息的更新操作。我们在第 9 章中曾经讨论过,这通常会利用上锁和开锁的信号量来加以实现。就像我们所看到的那样,由于操作系统可以协调上锁和开锁操作,所以这种机制在基于单处理器运行的系统中能够正常运作和发挥效用。然而,当有关进程是运行在彼此独立的系统上时,便没有哪个单一的操作系统可以来实施上锁和开锁操作。当前,针对分布式系统中的上锁和开锁操作主要设计开发了两种不同的方法,即**集中式锁定机制服务器**(centralized lock server)和**分布式锁定算法**。集中式锁定机制服务器相当简单,具体就是创建一个中央服务器,同时将所有上锁和开锁操作请求发送到该服务器上,如图 17-9 所示。该服务器采用先来先服务的原则加以运作和处理相关请求。

但是,集中式锁定机制服务器是故障要害(Single Point of Failure, SPoF)所在,同时也是潜在的性能瓶颈,因此有时采用的是**分布式锁定算法**,如图 17-10 所示。根据这一算法,希望进入临界区的进程将向所有其他进程发出关于锁定临界区的许可请求,并在有关请求中包含一个逻辑时钟时间戳。如果接收到请求的进程当前不想访问相应的临界区,那么它便可以

立即批准请求者进程的许可请求。但如果接收到请求的进程自身也想访问相应的临界区，那么它将会比较自己的时间戳与所传入请求的时间戳。如果所传入请求的时间戳较早，那么它也将批准请求者进程的许可请求。否则，它将在自己完成相应临界区的使用之后才会批准请求者进程的许可请求。在图 17-10 中，所有三个客户端都想在自己的进程中访问相关的临界区。客户端 3 向所有其他的客户端发出带有逻辑时钟值 157 的消息。客户端 1 自身的逻辑时钟值为 155，因此该客户端将会延迟对客户端 3 的许可请求的批准，直到其自身完成对临界区的访问。而客户端 2 将会直接批准客户端 3 的许可请求，因为来自客户端 3 的请求拥有比其自身更低的逻辑时钟值。（注意，该图中并未给出源自客户端 1 和客户端 2 的许可请求。）

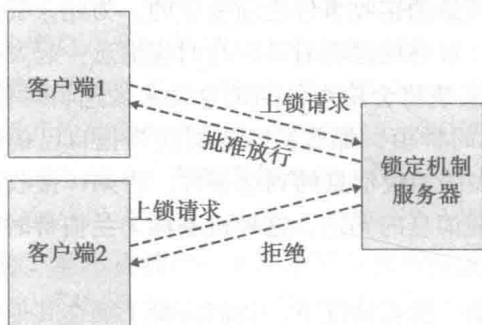


图 17-9 集中式锁定机制服务器

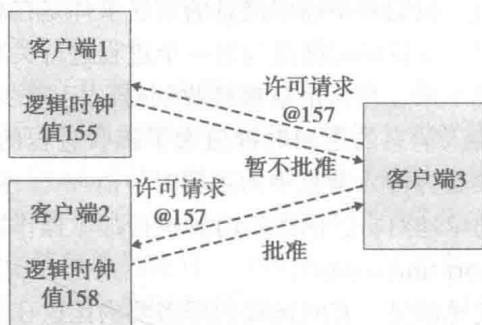


图 17-10 分布式锁定

17.6.3 选举

如果分布式系统正在使用集中式服务器的相关功能，譬如在上一节中所提到的锁定机制服务器功能，那么需要做出的设计决策所牵涉的一个问题是：假设任何客户端都可能提出有关操作请求的情况下，服务器应当如何确定和执行相应的操作。类似地，在某些算法中，我们往往会设立一个进程作为有关算法的协调器（coordinator）。在大多数情况下，服务器（或协调器）是由系统管理员所选定的，然后有关功能便运行在相应的服务器（或协调器）上面。但是，我们必须考虑到，协调器是故障要害所在。如果我们感兴趣的是一个总体上更为可靠的系统，那么我们有必要能够让有关功能运行在不止一处地方，从而预防主站点出现故障或无法访问的情况。为此，立足于最普适化和最可靠的考量来说，就应当是允许有关功能可以运行在任何地方。所以，我们需要动态地确定哪个进程应当运行有关功能。服务器或协调器进程的动态确定被称为选举（election）。而通常用于此类选举操作的两种算法是强者算法（bully algorithm）和令牌环算法（token ring algorithm）。无论是什么样的选举算法，所有节点均需分别拥有某个预先指定的（关于节点自己成为协调器的）优先级，并且有关算法应当选择具有最高优先级的进程作为协调器。

第一个必须要探讨的问题是一个相对简单的问题，即节点是如何决定需要一次选举的。一般而言，选举可能在三种情况下需要实施：当一个节点加入节点组的时候、当网络故障把整个网络分割开来从而使得相应节点组的部分节点无法连接到协调器的时候以及当协调器发生崩溃的情况下。当一个节点加入节点组时，它可能拥有关于自己成为协调器的最高优先级，因此在这种情况下，它将总是启动运行有关选举算法。对于其他两种情况来说，相关进程应该使用定时器来检测自身与协调器之间通信的缺失与故障问题。进一步说，如果定时器到期但却没有收到来自协调器的消息，那么相应进程将会启动选举操作。为此，这可能需要协调器在没有其他消息要发送的情况下向相应节点组所有成员发出“连接保持”的通知，从

而使得其他节点不会触发不必要的选举操作。

在**强者算法**中，每个进程均被指定了一个优先级，而且需要启动选举操作的进程将会向对应节点组中的所有其他进程发出消息，呈上它自己的优先级并公布自己想成为协调器的意愿。接下来，接收到该消息的任何进程，譬如 P，将会比较所传入消息中给定的优先级和进程 P 自身的优先级的大小。

- 如果所传入消息中的优先级高于进程 P 自身的优先级，那么接收者进程只是退出相应算法即可。
- 如果进程 P 自身的优先级高于所传入消息中的优先级，那么进程 P 将会给以消息答复，声明其自身的优先级，于是最初发送消息的那个进程从算法中退出。
- 最终，获胜进程将会向整个节点组发出广播消息，宣布其协调器身份。

我们看到，在图 17-11 中，有三个进程发出了关于强者算法的消息。假设优先级 1 为最高优先级，那么相应的进程将会在该算法实施过程中脱颖而出和赢得协调器身份。

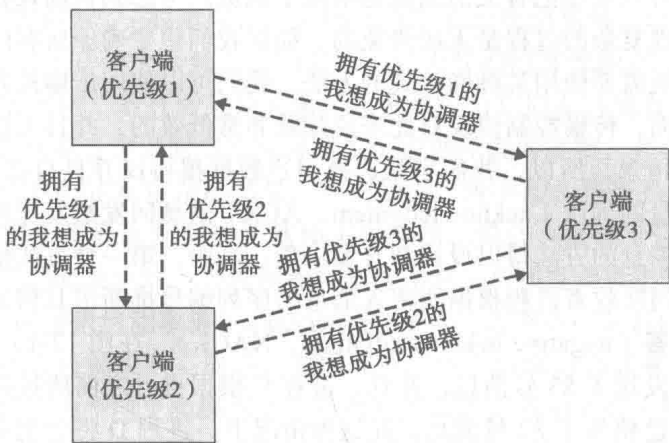


图 17-11 强者算法

另一种选举算法是**令牌环算法**。在该算法中，每个进程被设定了一个数，从而建立起了关于逻辑环中的一种次序。每个进程都需要了解整个环的次序。发起选举操作的进程将会向其所认为的逻辑环中的下一个进程发送一条包含其自身的进程号和优先级的选举性消息。如果它没有收到任何答复，那么它将会把这一消息发送给其所认为的逻辑环中的再下一个的进程。最终，肯定会有某个进程对此消息做出响应。该响应进程将会在此消息中附上其自身的进程号和优先级，并继续把该消息传向环中的下一个进程，其间同样也会绕过因发生故障而失败的任何进程。当这一消息最后返回到当初发起选举操作的进程时，该消息将会包含当前所有的进程及其对应优先级的一个有序表。然后，这一最终形成的消息将会再次在环中发送一遍。因此，每个进程将会获悉最终成为协调器的进程的进程号以及整个环的全部次序。相关过程如图 17-12 所示。其中，客户端 A 率先发出了选举性消息，而客户端 B 和客户端 C 在该消息在环中传递时依次向该消息附着上了自身的标识符和优先级。当该消息返回到客户端 A

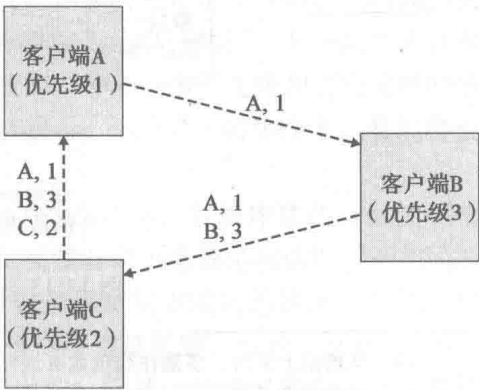


图 17-12 令牌环选举

时，客户端 A 将会获悉自己成为协调器。于是，这一消息将会在环中再次被发送一遍，从而使得所有进程都将知道整个节点组的成员资格以及对应于环中的次序。

17.6.4 可靠的多播通信

通常情况下，协作进程组要求实现组内各成员之间的可靠的通信。向组内的所有成员发送一条消息，并且不会把该消息发给任何其他的实体，这被称为多播[⊖]（multicast）。遗憾的是，除了单个网际协议网络内部的多播外，传输控制协议及网际协议并不支持多播机制，而介质访问控制层的多播也仅仅限于单一的局域网。相比之下，用户数据报协议倒是支持多播，但却并不可靠。因此，我们不得不想方设法在应用层探索更为可靠的多播方案。在所有各种情况下都可以正常运作和发挥效用的唯一机制是，协作进程组的每个成员均需拥有一条到达组内每一其他成员的点对点的连接。这在互联网上很容易就能够做到，尽管对于大型的协作进程组而言，其伸缩性并不强。于是，当一个进程想要向组内的所有成员发送一条消息时，它只是通过点对点连接把有关消息发送给每个成员。考虑到目前在较低网络层中的可用设施，故而相关繁琐复杂的过程是无法避免的。如果我们想要确保所有的进程都会看到所有的消息，那么我们就需要使用某种确认式的方法。我们可以使用传输控制协议，毕竟其内置了这样的保证。然而，传输控制协议对此来说却是非常低效的，并且无法很好地延伸和扩展应用到大量进程的情景。所以，我们宁愿选择用户数据报协议并且自己来实施确认。当然，这仍然必须得把大量的确认（acknowledgment, ACK）消息回发给发送者进程。因此，若干种最少化此类确认消息的方法得以设计和开发出来。其中，第一种方法依赖于当今网络的高可靠性，而且只有当接收者进程根据所传入消息的序列编号推断出其错失了某条消息时才会发送所谓的否定应答（negative acknowledgment, NACK）。在图 17-13 中，我们可以看到，进程 D 向其他进程发送了 83 号消息，并且，进程 C 根据自己之前所接收的最后一条消息是 81 号，从而获悉自己错失了 82 号消息。在这种情况下，进程 D 将会把对应丢失的消息重新发送给所有错失该条消息的接收者进程。当然，还有人提出了一些其他的改进方案，但是既未引起特别的注意，也未得到广泛有效的利用。

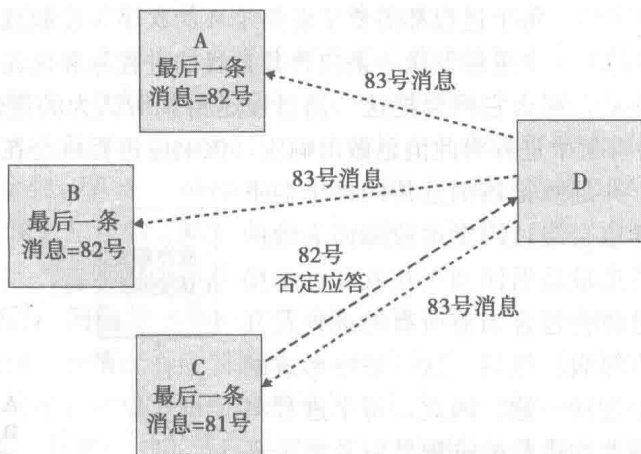


图 17-13 利用否定应答的可靠多播通信

⊖ 从理论上讲，多播往往包含有这样的思想，也就是说，有关消息只会向对应网络中发送一次，对应网络会把有关消息同时递送到所有目标位置，相关消息在对应网络的每条链路上只会递送一次，并且只有在到达目标位置的链路被断开时才会创建副本。不过，我们在这里却忽略了相关优化措施。

17.6.5 分布式事务

在交互式系统中，我们经常会处理各种类型的事务，而相关事务往往会牵涉数据库的若干更新操作。典型的例子，如关于把货品从一个仓库（A）迁移到另一个仓库（B）的库存事务。这通常需要完成如下几个步骤：

- 1) 读取仓库 A 中相应货品的数量。
- 2) 对仓库 A 相应货品的计数执行减 1 操作，并更新数据库中对应表项计数值。
- 3) 读取仓库 B 中相应货品的数量。
- 4) 对仓库 B 相应货品的计数执行加 1 操作，并更新数据库中对应表项计数值。

期间，如果系统碰巧在第 2 步完成之后且在第 4 步写入之前因发生了故障而陷入了崩溃，那么有关数据库将会呈现出一种**不一致的状态**（inconsistent state）。换句话说，我们将会遗漏掉库存中相应一件货品的跟踪记录。尽管我们可以通过按照相反的顺序来执行这两个更新操作，从而避免遗漏对任何货品的跟踪，但是那样，我们又会承担上“误以为我们额外添置了一件货品”的风险。一般来说，此类问题可以通过所谓**事务**（transaction）的机制来予以避免。“事务”似乎是一个重载性质的术语，因为我们时常使用这一词语来表示用户可能想要对系统完成的事情。不过话又说回来，相关过程与我们在这里看到的步骤非常相似。大多数的用户事务往往都会涉及对多个文件或多个数据库表的更新，并且我们希望整个系统可以正确无误地反映我们正在记录的结果。我们认为这种类型的更新是**原子**（atomic）性质的，也就是说，所有步骤要么全部被实施和正式存档，要么直接被全部丢弃，即便系统崩溃是在更新操作序列期间发生的。当进程要实施这样的一系列更新操作时，它会发起一个应用程序接口（API）调用来**启动事务**（transaction start）。而当每项更新信息写入数据库的时候，有可能成功，也有可能失败。故而，数据库系统将会以临时方式来执行这些更新操作，也就是说，采用一种认为相应更新操作是正在进行但尚未完成的方式来把对应更新结果写入数据库中。如果其间的任何更新操作被拒绝，那么相应进程将会发起**事务夭折**（transaction abort，或称为事务中止）的系统调用，于是数据库将会清除掉所有的临时更新结果。但如果所有的更新步骤均告成功，那么相应进程将会发起**事务提交**（transaction commit）的系统调用，故而数据库系统将会提交相关更新结果，实现持久更新，同时删除先前的任何临时记录。

在拥有分布式数据库的系统中，相关操作非常类似。但是，由于单个事务可能会牵涉若干不同的数据库服务器的协作，而从更新操作调用发起之后的过程当中，有可能其中的某个进程已经发生了故障，所以有关提交过程往往会略微复杂些。进一步说，其间，可能是管理分布式数据库相关要素的某一个进程发生了崩溃，或是相应网络发生了故障，故而我们无法与对应进程开展通信。无论是哪一种情况，有关操作都将无法继续执行。对于拥有非分布式数据库的系统来说，是不可能以这种分离的方式失败的。于是，为了支持和适应这样的情况，分布式事务处理便建立在所谓的**两阶段提交**（two-phase commit）的协议上。我们将会在下一节讨论有关协议。

关于分布式数据库相关事务的另一个问题是，为了提高性能，数据库往往会试图交错处理针对相同数据表的但来自不同进程的更新操作。只要没有两个进程是在试图更新相同的数据，那么便不存在任何冲突问题，故而相关操作可以采用交织方式向前推进。但如果有两个进程试图同时更新相同的数据，那么其中的一项操作应当被拒绝。注意，在这一点上，被拒绝的应用程序的最佳选择是简单地重新尝试相应操作。但是，这种情况在单处理系统（uniprocessing system）中不可能出现，因此其违背了分布的透明性。

17.7 容错

在我们关于透明性的讨论中，我们曾经说过，其中的一项目标就是让故障对用户是透明的。对于分布式应用程序容错能力的提升，可以采用许许多多的机制，不过，它们大多都与冗余密切相关。

17.7.1 概述

与单一系统中的故障相比，分布式系统中的故障是一个更为复杂的问题。当单一系统出现故障时，整个系统都将停下来。而当分布式系统发生故障时，其中只会有一部分可能停下来。鉴于相关部件彼此之间往往未必会不断地发生通信，所以，首要的问题就是让各种各样的部件设法推断和能够确定另一部件已经罢工和不再运作。但这件事情做起来却可能非常棘手。通常情况下，我们会从使用超时的想法出发来着手处理。当客户端请求服务器完成某项事情的时候，它往往会启动一个定时器。如果在某个确定的时间期限内没有得到答复，那么该客户端便可以推断相应的服务器已经宕机。然而，有关服务器可能并没有宕机，也可能是发生了网络问题使得服务器当前无法访问。譬如说，在网络中的某个节点处可能出现了通信流量的骤增，从而使得其间的路由器不得不丢弃掉了相应的请求或者应答结果。再比如说，也可能是当前服务器超载运行，尽管有关操作已经执行，但是相关应答结果却尚未发送或者仍在传输过程中。在这种情况下，如果我们重新尝试有关操作，反而可能会导致问题的发生。进一步说，如果有关操作是从库存计数中减去一件货品，我们肯定不希望重复这一操作。不过，话又说回来，如果我们只是统计库存数量，然后把计数变量设置为我们所获悉的仓库中所包含的货品数量值，那么重复此类请求不会造成任何问题，当然前提条件是期间没有发生过库存的其他更改。我们把后一类操作称作是**幂等的 (idempotent)**。具体而言，幂等性质的操作将以这样的一种方式执行存档，即其可以重复执行但不会改变结果。值得注意的是，如果我们将减法操作记录为旧值的读取和新值的写入，那么这也将是幂等性质的。

17.7.2 进程韧性

通过把相关功能分布到运行在不同系统上的多个进程上，我们就能够使这些功能更为健壮。也就是说，如果有某个进程发生了问题，其他进程可以继续正常运行。我们谈到这样的系统时，称其拥有**进程组 (process group)**。进程组可以组织成分层式进程组 (hierarchical group，如图 17-14 所示)，也可以组织成平坦型进程组 (flat group，如图 17-15 所示)。对于分层式进程组而言，会有一个进程作为协调器，而其他进程均需通过点对点链接方式向协调器进程提交报告。拥有唯一的一个协调器进程意味着该进程组将会存在单点故障要害问题，就像我们在上一节中所看到的那样，尽管我们可以选举产生新的协调器进程。但在选举期间（以及识别故障所需的超时期限过程中），相应进程组基本上是不起

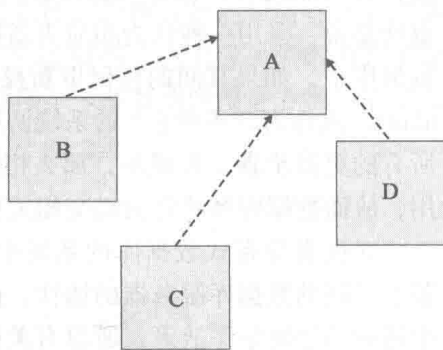


图 17-14 分层式进程组

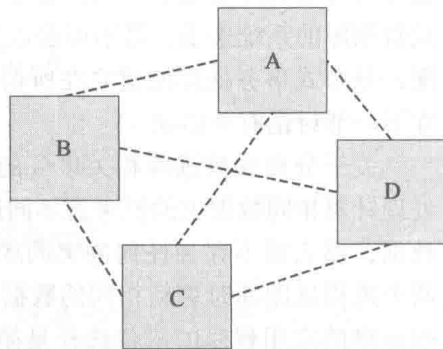


图 17-15 平坦型进程组

作用的，故而整个系统将会变得不太稳定。总体上来说，此类系统比较容易实现，并且相关通信开销也不大。

407

对于平坦型进程组而言，相关控制分布遍及整个进程组。每个进程与其他各进程之间直接进行通信。因此，相应的通信开销远远大于分层式进程组的通信开销。不过，这种进程组方式健壮性更强，因为单点故障不会导致整个进程组的失效。当然，这样的系统实现起来要比分层式进程组复杂许多。

17.7.3 可靠的客户端 – 服务器通信

前面我们曾经说过，如果服务器无法正常响应，那么客户端需要做出一些特殊的考虑和判断。譬如说，我们可能不希望重新发送有关请求，因为我们不想让有关操作再执行一遍。同样，如果服务器发生了崩溃，我们将会陷入左右为难的境地，因为我们并不真正清楚对应服务器是否已经接收到了有关请求、是否对请求进行了处理以及是否是在能够发出应答之前崩溃的。但是，有时候，根据有关请求采取行动却是非常重要的。或许我们可能正在发出火灾报警，不止一次地发出这样的报警并不是一个大的问题。这种情形被称为**至少一次性语义** (at least once semantic)。当然，也有一些时候，有关请求特别不希望被发送一次（不含一次）以上。就此不妨想象一下，从我们的银行账户支付一笔账单的请求！我们肯定不希望这种情况的发生超过一次。这种情形被称为**至多一次性语义** (at most once semantics)。对于其他的情况，我们或多或少抱持一种无关紧要的态度——譬如，对于发给证券报价机的请求，无非是列出股票的最新交易行情而已，多发一次请求或者少发一次请求都无所谓。这种情形被称为**不保证性语义** (no guarantee semantics)。最后，我们希望有关中间件最好能够保证事务恰好被处理一次 (exactly once)。对于中间件来讲，要达到这种级别的保证是有可能实现的，但是将会需要大量的日志和复核 (double-checking)，所以实现起来代价会比较昂贵。总的来说，我们必须针对每种类型的事务分别展开深入细致的分析，从而确认应当采取和授权什么级别的语义保证。

17.7.4 分布式提交

在前面一节中讨论分布式事务的时候，我们曾提到过一个分布式提交的概念。而在这里所讨论的有关算法则与故障透明性相关，所以是单独分开的，其被称为两阶段提交算法。具体而言，有一个进程将会作为整个算法的协调器。该协调器进程将会向其余每个进程发送一条消息，向那些进程征询它们是否可以提交相应所请求的更新结果。如果所有进程处于正常运行状态而且网络运作没有问题，那么相关进程都将返回肯定的答复，于是协调器接下来便会通知全体进程应当提交更新结果了。但如果有任何进程发生了故障或者网络不能正常运行，那么协调器将至少不会从一个进程那里接收到肯定的答复，因此这一消息应答过程最终将超过时限，故而协调器将会向其余每个进程发送一条中止请求。如果有进程发生了崩溃，还会引发一些复杂的问题。譬如说，当发生故障的进程重新启动时，它可能通过检查日志文件了解到其当时所处的状态（例如正处于提交操作的过程当中）。为此，相应进程接下来所需完成的事情将会取决于其在发生崩溃时所处的状态。如果它正处于中止状态，那么其仅仅需要中止相应操作即可。类似地，如果它正处于提交状态，那么它应当继续相应提交过程。以上这些状态只有在得到协调器相应指示的情况下才会进入。如果正在恢复的进程处于就绪状态，且正等待和监听来自于协调器的指示，那么它只需要请求协调器重复发送相应指令即可。

408

当协调器发生崩溃时，问题便比较严重了。在这种情况下，其他相关进程与协调器之间

的通信将会超时，从而使有关问题被暴露和发现。进一步说，如果有关进程正处于中止或提交状态，那么其接下来会按部就班地采取相应的行动。而如果有关进程正处于就绪状态，那么它是无法自己告诉自己去做什么事情的，于是它便会询问其他所有进程所处的状态情况。如果任何其他进程答复说正处于中止或提交状态，那么便意味着协调器在发生崩溃之前就已经做出决定并且已经启动发出了相应的指令，于是所有这些进程还是可以按部就班地采取相应的行动。

当然，也存在小概率的因为协调器在发送提交消息之前就已经崩溃故而有关算法可能挂起的情况。为此，设计和开发形成了这一算法的衍生版，即三阶段提交（three-phase commit）算法。但在实践过程当中，这种情况是非常罕见的，故而三阶段提交算法几乎从未真正使用过。

17.8 小结

分布式系统，如果不是已经，也将很快成为规范，而不再是特例或例外情况。本章首先回顾了关于为什么我们会发现分布式系统一天天愈加常见的一系列缘由。其间，我们还解释了分布透明性的概念，引入了中间件的思想，并阐明了中间件采取相应形式的理由所在。在此基础上，我们介绍了经常和分布式系统一起使用的几种不同的模型，包括客户端-服务器模型、三层模型、 N 层模型以及水平分布模型。接下来，在第17.3节，我们重温了进程和线程的相关原理，并就致力于客户端和服务器执行性能更好或者至少看起来更为高效的目标，如何在分布式系统中使用线程的方案进行了解释说明。鉴于分布式系统中的进程需要通信，为此，它们需要引用其他的实体。所以，我们在第17.4节中论述了命名和命名空间的概念。在第17.5节中，我们介绍了分布式系统的一些不同范例，包括远程过程调用、分布式对象、分布式文档以及分布式文件系统。与单一系统相比，分布式系统在同步方面拥有一些特殊的颇为不同的问题，所以我们紧接着讨论了分布式系统中的同步机制。第17.7节的内容主要围绕容错展开，这对于分布式系统来说同样存在一些特殊的问题，因为其中某个部件发生故障之后，系统的其余部分应当能够继续正常运行。

在本书的最后一部分中，我们将审视一些重要的现代操作系统，就那些我们在面向深度的主题章节中所描述的一些功能特征在相关操作系统中的实现机制展开进一步的分析讨论。其中某些操作系统曾经在第二部分中做过介绍，但其间我们仅仅讨论了相应操作系统为支持给定级别的功能特征所需拥有的功能类型。在本书的第六部分，我们将更为细致地对其中的一些操作系统进行深入的剖析，阐明它们是如何运用我们已经讨论过的有关机制的。

参考文献

- Barroso, L., J. Dean, and U. Hoelzle, "Web Search for a Planet: The Google Cluster Architecture," Research Paper, Google, Inc., 2005.
- Chandy, K. M., and J. Misra, "Distributed Deadlock Detection," *ACM Transactions on Computer Systems*, Vol. 1, No. 2, May 1983, pp. 144–156.
- Knapp, E., "Deadlock Detection in Distributed Databases," *ACM Computing Surveys*, Vol. 19, No. 4, December 1987, pp. 303–328.
- Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, July 1978, pp. 558–565.
- Obermarck, R., "Distributed Deadlock Detection Algorithm," *ACM Transactions on Database Systems*, Vol. 7, No. 2, June 1982, pp. 187–208.
- Ricart, G., and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Communications of the ACM*, Vol. 24, No. 1, January 1981, pp. 9–17.
- Rivest, R., A. Shamir, and L. Adleman, "On Digital Signatures and Public Key Cryptosystems," *Communications of the ACM*, Vol. 21, No. 2, February 1978, pp. 120–126.
- Sandberg, R., et al., "Design and Implementation of the Sun Network File System," *Proceedings of the USENIX 1985 Summer Conference*, June 1985, pp. 119–130.

网上资源

<http://www.opengroup.org/dce/> (开放软件基金会的
分布式计算环境 (DCE), 关于远程过程调
用的一种实现方案)
<http://www.w3.org> (万维网联盟 (World Wide Web

Consortium, W3C)
<http://www-306.ibm.com/software/lotus/> (其他分布式
产品, 包括 Lotus Notes 和 Symphony)
http://en.wikipedia.org/wiki/Two-phase_commit

习题

- 17.1 关于为什么分布式系统被发现越来越常见的原因, 我们在本章共列出了 8 条理由。请说出这 8 条理由中的 4 条。
- 17.2 我们列出了分布式系统在理想情况下应当对用户透明的 9 个方面。请给出其中的 5 个方面。
- 17.3 简要描述中间件的定义。
- 17.4 我们介绍了用于构建分布式系统的 4 种模型。其中哪种模型是万维网的基础?
- 17.5 添加到万维网模型中从而派生形成三层模型的那一层是什么?
- 17.6 根据谷歌的理解, 分布式系统模型的进一步泛化结果称为什么?
- 17.7 本章还描述了一种称为水平分布的不同类型的模型。请举例说明利用这种模型所描述的系统类型。
- 17.8 早期使用 1.0 版超文本传输协议的网络浏览器必须得打开一个单独的连接才能够检索网页上所引用的各个组件。是采用了一些什么技术才使有关操作变得更为高效的?
- 17.9 请描述通常情况下服务器使用线程的两种方式。
- 17.10 当物理虚拟机上的主机操作系统加载客户机操作系统时, 它是如何确保主机操作系统保持享有对整个系统的控制权的?
- 17.11 简要描述抽象虚拟机的操作机理。
- 17.12 对于 Jini 设计而言, 无须事先了解有关服务所使用的网络机制, 便可允许应用程序访问相应的服务。这是否正确?
- 17.13 关于为什么移动实体使用网际协议会成为问题的基本原因是什么?
- 17.14 关于我们所审视的中间件的一项应用, 就是支持我们利用现有的程序, 并将其中的部分模块迁移到另一个系统上。为做到这一点, 相应的非面向对象的设计方案是什么?
- 17.15 我们把开发分布式对象系统的主要标准称为什么?
- 17.16 在接收消息时, 兰伯特时间戳机制是如何确保本地逻辑时钟正确反映关于分布式系统中事件发生顺序的相关信息的?
- 17.17 集中式机制是如何运作和支持分布式系统中的互斥的?
- 17.18 用于选举一个系统的协调器进程的两种不同的分布式算法是什么?
- 17.19 传输控制协议支持互联网上的可靠的多播通信。这是否正确?
- 17.20 为什么分布式系统中的数据库事务更加难以处理?

实例研究

本书的前两部分为我们阐释了一些最初的背景，并按照我们所称的“螺旋式方法”介绍了一系列较为复杂的操作系统。采用螺旋式方法的目的在于凸显所探讨的功能特征的动机，并对相关素材给以更为客观的认识。接下来的三个部分则深入地解释了操作系统所涉及的各种各样的技术内容。在这一部分，我们将以实例研究的形式再次转向真实的操作系统。我们将会更加细致深入地阐述几种现代操作系统是如何整合和实现第三~五部分中所描述的相关功能的。

第18章涵盖 Windows NT 系列操作系统，从其第一个发行版本开始，一直到现在称为 Vista 的版本。其间还包含一些相关历史材料以为学生提供较为客观的审查视角。本章中的其他主题则分别讨论了单用户操作系统环境、进程调度、内存管理、文件支持、基本输入/输出、图形化用户接口编程、网络、对称多处理，以及后续版本提升启动速度的重要意义和关于 Vista 版本中新的功能特征的一些点评。

第19章围绕 Linux 操作系统展开，涵盖了教材第二部分并未触及的其他主题，以及该系统是如何实现我们期望在任何现代操作系统中所看到的一些标准功能的。在简要回顾 Linux 操作系统之后，我们讨论了 Linux 系统的内存管理功能以及文件系统组织。这一章还介绍了 Linux 系统的基本输入/输出功能、对图形化用户接口编程的支持、对网络的支持以及对称多处理机制。在此基础上，我们还介绍了由 Linux 衍生形成的一些很有意思的操作系统，主要是硬实时系统。

第20章讨论了 Palm 操作系统的其他主题，主要包括该操作系统在螺旋式方法部分未曾提及的其他很有意思的功能、与此类系统打交道时所需要的编程环境、手机市场的类似开发以及它们与掌上电脑市场所形成的对比。最后，该章还讨论了正在为这些操作系统所开发的新型应用程序，其间牵涉移动平台特性以及对操作系统功能特征的影响。

从 Windows NT 到 Windows Vista

在这一章，我们将讨论一个从安装数量来说占据绝对第一位置的个人计算机操作系统家族，即由微软公司开发的 Windows NT 系列操作系统。对于普通的观察人员来说，该操作系统似乎每次只能支持一个用户来使用个人计算机的控制台。然而，该操作系统实际上可以支持远程终端上的多个用户。它还通过运行各种远程访问的功能（如文件、打印和目录服务）来支持多个并发用户，同时也可用作诸如数据库、超文本传输协议服务器或网站服务器、文件传输协议服务器、网络服务以及许多其他高级服务的运行平台。在更高版本中，该操作系统还支持所谓快速用户切换的功能。这一功能允许一个用户从系统中注销，而正在运行的任何应用程序都将继续保留在内存中。然后，另一个用户就可以登录并启动运行另外的应用程序。同样，第二个用户也可以注销，同时让所有的应用程序保持运行。这时候，第一个用户可以重新登录返回系统并从上次停下的地方继续工作，其间并不需要重新启动相应的那些应用程序。

415

虽然本章的标题直接提到了 Vista，但我们采用术语 Windows NT 来指代相应整个操作系统系列产品。严格讲，这一术语仅仅适用于 3.1、3.5、3.51 和 4.0 发行版本的 NT 操作系统。尽管如此，有关产品系列命名不尽相同，而就绝大多数的 Windows 操作系统而言，相关发行版本之间的差异对我们来说并不重要。另外，术语 NT 通常用于指代相应整个版本系列，故而我们也将按这种方式来使用它。而如果我们提到在特定版本中某些功能特征的删减或增加，我们就可能说明特定版本的产品名称。

在本章的开始部分，我们将概要介绍 NT 操作系统以及各种各样的 Windows 操作系统发展历史相关的一些背景，以便提炼出关于各种功能特征和设计决策的某些观点。在第 18.2 节中，我们讨论了典型的 NT 操作系统环境的本质所在，同时还就 NT 操作系统的主要目标——多硬件平台的支持和传统 Windows 操作系统应用程序的支持进行了讨论。

NT 操作系统支持许多同时运作的用户进程以及并发执行的服务器功能，因此在第 18.3 节中，我们讨论了 NT 操作系统中的进程和任务的调度。NT 操作系统利用辅助存储器作为主存储器的扩展，因此需要复杂的内存处理机制。这些将会在第 18.4 节中加以讨论。同时，支持诸多服务器功能和多个用户的操作系统需要提供具有文件安全特性（就像 Linux 操作系统中所见到的那样）的复杂文件系统，所以，在第 18.5 节便就 NT 操作系统中文件的组织和结构以及文件系统元数据等进行了解释说明。而在第 18.6 节，则就 NT 操作系统所提供的用于支持相关更高级别功能的基础输入/输出功能予以阐述。

就像 Mac 操作系统和 Linux 操作系统一样，NT 操作系统图形化用户界面也支持多个重叠的窗口，故而需要一套精心设计的服务于图形化用户界面的应用编程接口。所以，在第 18.7 节中讨论了利用 NT 操作系统进行图形化用户接口程序设计的一些要领。运行 Palm 等操作系统的掌上电脑往往拥有精巧的通信选项，但在大多数情况下，它们每次只会使用一个。相比之下，在 NT 操作系统中，用户可能会同时开展许多通信活动，譬如查收电子邮件、通过互联网玩游戏、与掌上电脑进行数据库的同步，等等。为此，在第 18.8 节中，主

要就 Windows NT 操作系统中的各种网络支持进行了讨论。另外, NT 操作系统常常运行在具有多个处理器的系统上, 特别是当其主要用作服务器而不再仅仅是作为工作站的情况下。故而在第 18.9 节就 NT 操作系统对此类系统的支持方式进行了介绍。在第 18.10 节中, 我们还就 XP 发行版本的 NT 操作系统的启动速度提升的目的以及相关速度提升为什么重要的原因予以概述。最后, 我们在第 18.11 节中对整章内容进行了归纳总结。

18.1 Windows NT 系列操作系统发展历程

最初的某些发展历程: 如前所述, Windows 操作系统起初是为支持单个用户使用个人计算机而开发的。有关支持可以追溯到 8088/8086 处理器时代。微软在 1981 年开始研发支持图形化用户界面 (Graphical User Interface, GUI, 又称为图形化用户接口) 的操作系统。当时, 相应的操作系统被称为界面管理器 (Interface Manager, IM)。期间所使用的处理器在保护一个进程免受其他进程的影响方面缺乏必要的功能支持。由于这类硬件限制问题, 在此之前的大多数个人计算机操作系统均非多处理系统, 而且也不是界面管理器。尽管可以同时打开多个应用程序, 但实际上只会有一个程序处于运行状态。当时的窗口不能重叠, 而只能按平铺方式进行处理。平铺的窗口甚至连部分地覆盖彼此的情况都不会出现, 因此对于操作系统来说, 窗口管理非常简单。到了 1983 年正式发布界面管理器的时候, 相应操作系统的名称调整为了 Windows。就像技术开发领域经常发生的那样, 图形化用户界面的思想在好多机构同时得到了推进和发展。有关想法萌芽于施乐帕洛阿尔托研究中心 (Xerox Palo Alto Research Center, PARC), 故而 Windows 并非面向英特尔系列处理器的第一个使用图形化用户界面的操作系统。在 Windows 发布之前, 个人软件公司 (Personal Software, 后来改名为 VisiCorp) 就曾发布了 VisiOn, 该系统实际上是以类似于 UNIX 操作系统中的 X-Windows 的方式而运行在操作系统之上的一种支撑环境。IBM 也曾着手开展所谓 TopView 的多处理 8x86 环境的研发, 不过该系统并不包含图形化用户界面。

[416]

第一版的 Windows 操作系统只能算得上勉强成功, 这主要是因为硬件体系结构的限制和处理器的速度。以后的版本则利用了 80286 处理器中的更先进的功能, 从而为内存管理提供了更好的支持, 但是在图形显示方面, 这一时期的个人计算机的性能仍然显得有点力不从心。此外, 这些版本的 Windows 系统实际上是运行在原先的 16 位 DOS 操作系统上的外壳。而且, 它们大部分代码甚至完完全全都是用汇编语言编写的, 故而越来越难以增强, 甚至后来连维护都很困难了。在 80386 处理器上市的时候, 微软发布了一版称为 Windows 3.0 的操作系统, 并取得了极大的成功。尽管如此, 由于这时的 Windows 系统还是建立在 DOS 操作系统之上, 所以仍然存在大量的问题。这一产品拥有各种各样的迭代版本, 包括 Windows 3.1 和 Windows for Workgroups (面向工作组的 Windows 操作系统)。另外, 当时的 Windows 操作系统还存在硬件的指令集和寻址空间仅允许按照 16 位内存寻址空间的方式来进行设计等限制或缺点。再后来, 32 位指令集和内存模型的使用等实质性提高完善方案陆陆续续地融入了改进版的 Windows 系列操作系统中, 具体包括 Windows 95、Windows 98、Windows 98 SE (第二版的 Windows 98) 和 Windows ME (Millennium Edition, 即千禧版 Windows 操作系统) 等。

在开发早期版本的 Windows 系统的同时, 微软还参与开发了一个与 IBM 公司的 OS/2 相类似的操作系统。起初, OS/2 被看作是同时运行多个基于文本的程序的一种手段。随后, 拥有图形化用户界面并运行在 80286 和 80386 处理器上的其他版本的 OS/2 也陆续发布和发

行。在某种意义上，他们认定用汇编语言编写操作系统（就像 DOS 一样）并不是一个很好的主意，所以 OS/2 的大部分代码是用 C 语言编写的。起初，OS/2 所拥有的应用程序接口其实就是 DOS 系统的应用程序接口的扩展。第三版的 OS/2 由微软发起，并采用 OS/2 自身原生的应用程序接口对整个操作系统进行了彻彻底底的重写。然而，Windows 3.x 的巨大成功导致微软重新评估其最初的研发方向。于是，这一原生和自主设计的应用程序接口被进一步改造成面向 Windows 95 及更高版本而开发的 32 位 Win32 接口。从某种程度说，这种变化的结局就是，IBM 公司和微软在 OS/2 的研发合作方面分道扬镳。IBM 公司自己留下来继续开发 OS/2，而微软则把有关发行版的名称改成了 NT。

除了 Win32 应用程序接口，NT 也拟定要支持面向 DOS 系统和 Windows 3.x 系统而开发的 16 位应用程序。此外，许多美国政府和公司采购都要求 UNIX 风格的应用程序接口。因此，NT 还包括对按照 UNIX 系统标准化的 POSIX.1 API 所编写的应用程序的支持。（尽管 POSIX 接口实际上是独立于特定操作系统的，但是其得到了各个分散的 UNIX 社区的推动，并在很大程度上依赖于相应的应用程序接口。）

417

在当时，英特尔 x86 处理器（Intel x86）尚未在个人计算机领域取得目前这种主导支配地位。如果有哪家公司的处理器主导了个人计算机市场，那么微软便需要确保他们的操作系统能够运行在对应的处理器平台上。但如果没有哪家处理器占据主导地位，那么他们的操作系统就需要运行在大多数的处理器甚至是所有的处理器平台上。因此，他们把可移植性（portability）确定为其主要操作系统产品的首要目标。这意味着他们不得不抛弃基于 DOS 操作系统的 Windows 产品，而去编写一种新的操作系统。为了确保可移植性，他们决定采用高级语言来编写这种新的操作系统。毋庸置疑，采用高级语言还有许多其他的理由。为了构建这种新的操作系统，他们聘请了一群经验丰富的操作系统设计人员。他们原先打算采用 C++ 语言来编写操作系统，且最初把英特尔 i860 处理器等作为目标运行平台。英特尔 i860 处理器是一款精简指令集计算机（Reduced Instruction Set Computer, RISC）处理器。有关团队所使用的那款处理器芯片称为 N10，且微软使用的是所谓 N10（N-Ten）的 i860 仿真器。这就导致了 NT 名称的诞生，同时也寓意新技术（New Technology）。需要说明的是，有关硬件后来被证实在支持面向对象编程（object-oriented programming）方面颇显乏力，所以 NT 操作系统的内核最后几乎完全是采用标准 C 语言编写的。

各种各样的 Windows 产品和微软的早期系统之间的最大区别体现在图形化用户界面上。显然，今天的图形化用户界面非常普遍——这可能需要稍加解释一下，毕竟当 Windows 操作系统刚刚创建时，图形化用户界面对微软的操作系统而言尚算是新鲜玩意。此类界面极大地增强了用户体验，扩展了用户与系统交互的方式，可以说是远远超越了面向文本的终端提供的相关方式，并且还增加了一次运行多个任务的能力。多个程序运行在单独但可能重叠的基于图形的任务窗口中，同时通过诸如鼠标或触摸板（touch pad）等指针式设备（pointing device）在屏幕上移动一个指示器来加以控制，这种结合取得了巨大的成功。今天，除了嵌入在家用电器和其他机器中的系统之外，很少有操作系统不包含此类界面。需要强调的是，在诸如 UNIX、Linux 和 Mac OS X 等操作系统中，图形化用户界面是位于操作系统顶部的单独的一层。但在微软的 Windows 产品中，图形化用户界面则是操作系统设计的一个有机的组成部分，并且，自从至少是 Windows 2000 发行版以来，图形化用户界面一直是内核的一部分。

早期的 NT 操作系统的工作有时是在 MIPS 架构（Microprocessor without Interlocked

Piped Stage, 无锁管道阶段的微处理器)的系统上完成的。后来,微软终于下定决心,他们希望用 NT 系统取代所有现有的 DOS 系统和 Windows 系统,故而增加了针对 80x86 系列处理器的支持,并由于关系到一般的操作系统使用的芯片问题而最终放弃了 i860 处理器。另外还增加了对其他处理器的支持,包括 DEC Alpha 64 位处理器、MIPS RISC 处理器以及 PowerPC。(MIPS 芯片用于多个机器系列中,包括硅谷图形公司工作站。)然而,市场最终对这三种处理器在个人计算机中的使用投下了反对票,所以它们在后来的 NT 操作系统版本中不再给予支持。与此同时,在 XP 版的 NT 操作系统中,又增加了对英特尔安腾 64 位精简指令集计算机处理器以及英特尔公司和 AMD 公司的 64 位 x64 系列处理器的支持。因此,硬件平台独立性的理念一直是 NT 系列操作系统的一项重要特征。

NT 系列操作系统包括 Windows NT 3.1、3.5、3.51 和 4.0, Windows 2000(内核版本 NT 5.0), Windows XP(内核版本 NT 5.1), Windows Server 2003, Windows XP x64 版,以及现在的 Windows Vista(内核版本 6.0)。NT 操作系统产品系列包括对图形化用户界面、虚拟内存、日志式文件系统(Journaling File System, JFS)、抢占式多任务(preemptive multitasking)以及一整套网络协议的支持。总的来说,NT 系统是一种非常高端的操作系统。虽然在此期间,整个产品系列进行过一些显著的增强和改良,但我们在本章中描述的系统体系结构所涉及的大部分内容自从 NT 操作系统第一版发行以来就基本上没有发生过变化。

418

Windows Vista

Windows Vista 是最新版的 NT 操作系统。关于 Vista,微软的主要目标是提高 NT 操作系统的安全性,不过同时也在许多其他方面进行了改进和增强。在此,我们简要阐释 Vista 版操作系统的一些功能特征,以具体说明当前操作系统研发过程中所实施的各种举措。换句话说,其中的许多功能在其他的现代操作系统中也可以找到。Vista 系统中与 NTFS 完全相关的功能特征将在第 18.5.5 节中展开讨论。与安全性或可靠性密切相关的几项新的功能如下所述:

- 代码完整性验证(code integrity verification)。目前,操作系统加载程序和内核会针对所有的内核模式的二进制代码执行加载时的检查,以验证磁盘上的有关模块没有发生过更改。这将有助于防止恶意程序(malicious program)通过修改操作系统来控制机器。
- 服务安全提升(service security improvement)。现在,各种服务可以明确标定它们所需要的特权(例如,关机、审计、受限写等),从而约束有关服务的权限。未予明确指定的特权将被全部去除,故而限制了受损服务可能对操作系统所造成的损害。
- 用户账户控制(User Account Control, UAC)。在管理员授权提升特权级别之前,应用程序均被限制为标准用户权限,通过这样的用户账户控制策略将可以有效地改进相关安全特性。进一步说,一个用户可能拥有管理员权限,但是该用户所运行的应用程序通常仅具有标准用户权限,除非事先得到批准或该用户明确授权其具有较高权限。用户账户控制功能将自动提示用户提供额外特权,或者用户也可以右击程序图标,然后选择“以管理员身份运行”,从而提升特权级别。
- 地址空间布局随机化(Address Space Layout Randomization, ASLR)。这是一种通过对进程的各部分地址空间进行随机分配从而改善相关安全的技术。地址空间布局随机化技术通常会牵涉可执行程序的基址以及库、堆空间(heap space)和栈空间(stack space)。有关机制将会通过防止攻击者对其所设定的攻击目标的相关组件地址

的预测来阻止某些类型的安全攻击。

- 用户模式驱动程序框架 (User-Mode Driver Framework, UMDf)。大多数驱动程序都运行在内核模式下, 并对物理地址空间和系统数据结构拥有完全的访问权。此类访问有可能使恶意驱动程序或编码存在问题的驱动程序诱发某些问题, 进而影响到其他驱动程序或系统本身, 并最终造成整台机器的崩溃。相比之下, 在用户模式下运行的驱动程序只能访问用户地址空间, 存在的风险要小得多。Vista 新增了对这种用户模式驱动程序的支持。用户模式驱动程序框架是专为像照相机和便携式音乐播放器等设备而设计的。

其他一些与可靠性有关的新功能还包括:

- Windows 错误报告机制。这项功能用于捕获源自于终端用户那里的应用软件崩溃或挂起等相关的数据 (在征得有关用户同意报告的前提条件下)。软件开发人员可以在线访问与其应用程序相关的数据, 监控错误趋势, 并下载相关调试信息。
- 可靠的睡眠状态。在以往, 应用程序或驱动程序可能会阻止系统进入睡眠或休眠模式 (hibernate mode, 是一种睡眠状态)。由此带来的问题是, 笔记本电脑用户常常不会意识到自己的系统没有进入睡眠状态, 故而最终会导致放在包里的笔记本电脑变得过热、电池耗尽而数据丢失的后果。相比之前, Vista 系统在进入睡眠状态之前不会询问进程, 并且把用户模式下的超时通知从 20 秒减少到了 2 秒。
- 清理服务后再关机。在 Vista 之前的 NT 系统, 有关服务是没有办法延迟关机时间的。换句话说, 在事先确定的超时期限到达的情况下, 尽管相关服务还在继续运行, 但整个系统也会立即关机。因此, 对于那些需要把数据刷新存放到磁盘的服务来说, 这便可能导致某些后果。利用 Vista 系统, 如果有服务请求挂起关机操作, 那么该服务便可以根据需要来延迟关机时间。实际上, 在关机时, 首先会由通知型服务向所有处于运行状态的服务发出通知, 然后系统将等待这些服务停止工作。在此期间, 有需要的服务便可以请求挂起关机操作。当有关服务全部停下来后, 系统才会继续正常的关机操作。
- 服务关闭排序机制。Vista 系统支持针对服务指定一种关闭次序, 即服务关闭时需要遵循的相关服务间的依存关系。

此外, Vista 系统还新增了一些功能, 主要是着眼于改善总体性能和缩减关机或重启系统所需的时间:

- 延迟式自动启动服务。在 NT 系统中运行的服务通常被设置为自动启动类型, 因为稍后可能就会需要这些服务。然而, 相关服务一直在不断增多故而使得系统启动所耗费的时间也在持续延长。但是, 许多自动启动类型的服务不一定非得在系统启动时就立即启动, 它们仅仅需要的是一种无人参与的启动方式, 所以在系统启动后, 这些服务再启动准备就绪也完全可以。为此, Vista 提供了一种称为延迟式自动启动的新的选项。被指定为延迟式自动启动的服务将会在系统启动后不久再予以启动, 从而加快了系统启动过程, 并改善了用户登录体验。
- 超级预提取机制 (SuperFetch) 通过分析经常使用的应用程序, 并试图把那些常常访问的应用程序保持在主存中, 从而使它们能够更加快速地启动执行。该机制还将关注什么时候相关预取的数据会被移出到页面文件 (page file), 并且将会对导致预取数据移出到页面文件的应用程序实时监控。一旦有关应用程序执行完成, 超级预提取

机制将会把对应预提取的数据再次拖回到内存。而当用户再次访问相关的应用程序时，其所需要的预提取的数据就已被再次存放在了主存中。

- 闪速缓存机制 (ReadyBoost) 用于在闪存设备上创建缓存内存。尽管闪存设备的数据传输速度要低于当前的硬盘驱动器，但是闪存设备既没有寻道时间开销，也没有旋转延迟时间开销，因此它们可以大幅度提高硬盘的表观速度 (apparent speed)。请注意，这与我们在第 14 章关于将来替代旋转式存储器的预测是一致的。
- 闪速驱动器 (ReadyDrive) 机制用于支持有关的混合硬盘驱动器。(hybrid hard drive) 作为一种新型驱动器，混合硬盘包含有大容量的闪存缓冲区。当数据在主存和硬盘驱动器中的闪存内存 (flash RAM) 之间进行迁移的时候，驱动器在大部分时间可以处于断电状态，所以这种硬盘可以显著降低驱动器的功耗。与此同时，鉴于相关部件的移动操作大量减少，所以此类驱动器的可靠性得到了增强。另外，由于从闪存上读取数据要比首先等待盘片旋转然后查找数据的过程快出许多，所以整个系统将会拥有更快捷的启动速度。

420

18.2 用户操作系统环境

因为 NT 系统环境从根本上来说就是图形化用户界面，所以用户可以轻松地在屏幕上打开许多窗口并启动运行多个应用程序。对于 NT 用户来说，在任何时候让十几个甚至更多的应用程序运行，这也是稀疏平常的事情。一般而言，往往会有一个电子邮件阅读器在不时地查收所传入的邮件；同时一个约会日程安排程序也已打开；而网络浏览器，则可能打开了一个门户页面，上面正“播报”着最新新闻，并不断刷新相应用户股票投资组合的统计数据；用户呢，正在电子制表软件或者其他的办公类型的应用程序上工作；有一个窗口则显示着用户刚刚查找过单词的电子词典；另外还有一个即时通信 (instant messenger) 应用程序也在运行中。这些还没有包括可能正在运行的许许多多的其他的实用程序，例如本地防火墙、剪贴板编辑器、电池状态指示器、音量调节面板等等。与此同时，还可能运行着像共享打印、个人网页服务等服务器功能。因此，尽管 NT 操作系统被许多用户看作是单用户系统，但这绝不意味着该操作系统仅仅只有少数几样东西正在运行中。

目标：跨多硬件平台及操作系统平台模拟

跨越多种硬件平台的可移植性以及来自传统操作系统的应用程序的支持是 NT 操作系统开发人员的两项主要目标。为了实现第一项目标，微软运用了两种方法。首先，在很大程度上，操作系统内核的某些低级的硬件相关的部分被分离出来并被组织放置在所谓**硬件抽象层** (Hardware Abstraction Layer, HAL) 的单一模块中。当然，其他模块也会拥有部分代码对硬件存在依赖关系，例如，内存管理模块必须得知道物理内存的页面大小。但是，通过硬件抽象层的建立及部分地把操作系统中依赖于硬件的代码隔离在单个模块中，便使得将整个系统移植到新的硬件平台的过程大大简化了。需要说明的是，硬件抽象层往往随着与处理器一起使用的支持芯片 (例如，中断控制器)、系统是单处理器还是多处理器系统以及基本输入/输出系统支持的是什么样的电源管理特性等因素而发生改变。同时，有关芯片把总线和其他设备连接到处理器上，故而有时也会规定某些特定的指令，就像处理器一样。第二种技术就是采用与机器无关的更高级的语言来编写操作系统的所有其余部分。最初选定的语言是 C++，其初衷是让整个系统成为彻彻底底的面向对象的系统。然而后来，这一策略进行了调

整，相关限制采取了宽松处理，进而出于效率的考量，大量的 NT 代码都利用 C 语言进行了编写。NT 操作系统无须进行重大的改写就能够支持若干不同的处理器的事实表明，其在这方面的目标已得到非常成功的落实。

至于第二项目标，即正确高效地运行传统应用程序，也已基本实现。由于 NT 操作系统强力实施只有操作系统才被允许直接控制硬件的限制，所以有许多 DOS 系统和少数 Windows 3.x 系统的应用程序因直接使用了硬件而将无法在 NT 环境中运行。大多数的没有直接操作硬件的应用程序将会在 NT 环境中正常运行。支持传统应用程序的关键设想是在内核之上另外增加一层，且这一层通过把传统的应用程序接口调用转换为原生的 NT 操作系统的应用程序接口调用来实现对传统的应用程序接口调用的支持。在 NT 系列操作系统中，这些额外的层次被称为“环境”(environment)或“子系统”(subsystem)。事实上，即使被认为是 NT 系列操作系统标准的 32 位 Windows 应用程序接口也不是 NT 系统内核自身原生的应用程序接口。相关子系统如图 18-1 所示。到 XP 操作系统发布时为止，全世界的计算机系统基本上已抛弃了 OS/2 操作系统，故而 XP 版的 NT 操作系统就停止了对 OS/2 子系统的支持。另外，起初 NT 操作系统所包含的 POSIX 支持仅仅是 IEEE 1003.1/ISO 9945-1 标准的最小限度的实现，其在 XP 版操作系统中先是被撤销，随后被更完整的实现方案所取代。

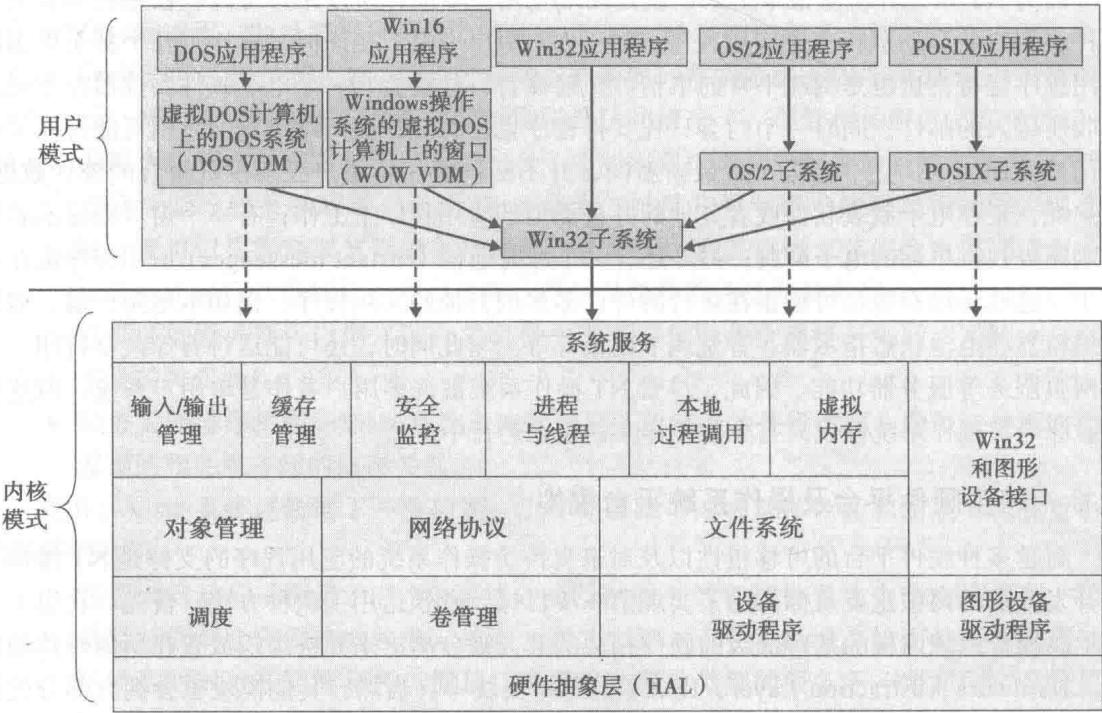


图 18-1 最初的 Windows NT 系列操作系统体系结构

可以肯定的是，NT 系列操作系统起初还有许多其他的设计目标，譬如性能、可靠性以及关于构建一流操作系统的高级目标，这与受到有限资源羁绊的早期版本的 Windows 操作系统存在很大的不同。但是，可移植性和兼容性的目标则可能是对整个系统设计影响最大的目标。

有关子系统并不都是简单和直接的，运行以前的应用程序有时会导致很多问题。例如，DOS 操作系统是单用户系统，因此应用程序通常会启动输入 / 输出操作，然后就执行自旋锁以等待相应操作完成。而 NT 操作系统对处理器进行了虚拟化处理，所以其他的应用程序不

会被锁定，但是在此期间 DOS 应用程序可能消耗掉大量的处理器周期。类似地，默认情况下，所有 16 位的 Windows 应用程序是以在 Windows 操作系统的虚拟 DOS 计算机上的窗口 (Windows on Windows Virtual DOS Machine, WOW VDM) 中的线程方式来运行的。按照线程的调度方式，如果有一个 Windows 应用程序停止接收输入，那么所有那些应用程序都将被挂起。

18.3 进程调度

NT 操作系统使用一种复杂的机制来控制 and 调度当前正在运行的进程。进一步说，该系统往往会启动运行多个进程，并为每个进程创建至少一个线程。在此基础上，它将会调度有关线程而不是进程来加以执行，而且其间所使用的是多级反馈队列 (multilevel feedback queue) 机制。NT 系统中的每个线程均会拥有一个取值在 0~31 之间的优先级，以告知操作系统相应进程在尽快被调度运行方面的重要程度。线程优先级根据对应进程的基本优先级 (base priority, 将在下面给出定义) 派生形成。对于这 32 种优先级中的每一个优先级别，都会存在一个单独的队列来组织那些已为运行准备就绪的线程。当一个线程启动运行时，其被赋予一个限定的时间量 (time quantum, 或称为时间片, 即 time slice) 来加以执行。当对应时间期限到达时，该线程将会被挂起，并被放置在其优先级对应的运行队列的末尾，同时该优先级的下一个线程将会被启动运行。对于每个优先级别而言，只有当在该优先级别上为运行准备就绪的所有线程都已执行完后，调度器才会移到下一个较低优先级的队列上进行调度。如果一个线程正在等待的某个事件 (例如等待磁盘读取一些数据) 已经完成和发来信号，那么操作系统将会检查等待对应事件的线程是否拥有比当前正在运行的线程更高的优先级。如果是，那么当前线程将会被挂起，并且该较高优先级的等待线程将会被启动运行。正如在第 8 章所论述的那样，因时间片用完和因高优先级事件而被中断的线程都是抢占式多任务 (preemptive multitasking) 的例子。

当一个进程启动时，应当为该进程确定一个初始的基本优先级类别 (base priority class)，如图 18-2 所示。该类别用于确定相应进程中的所有线程的基本优先级。而当进程中的有关线程执行的时候，它们的优先级可能会根据它们执行的操作而发生改变，这被称为动态优先级 (dynamic priority)。线程优先级的降低或提升是有一定限度的，也就是说，不能降到某一优先级别之下，也不能升到某一优先级别之上。伴随线程运行时的优先级的这种改变就是相关机制名称“多级反馈队列”中所指的“反馈”的内涵所在。而这样的提升和降低优先级的目的则在于，赋予那些密集聚焦在用户界面上的交互式进程以更高的优先级，同时适当降低那些后台的看起来较少介入用户界面的进程的优先级。

因此，对于诸如键盘输入之类的交互式任务所涉及的线程来说，NT 调度器赋予了它们较高的优先级。为了做到一点，NT 调度器使用了一种与第 8 章中所论述内容略有区别的机制。如下为 NT 系统提升线程优先级的几种情况：

- 当一个线程执行了阻塞式调用并且相应请求已经完成时，该线程的动态优先级应当被提升，以便它可以充分利用对应已经完成的执行。
- 当与普通 (NORMAL) 优先级上的一个进程相关联的窗口获得焦点时，调度器应当提升对应进程的优先级，使其大于或等于所有后台进程的优先级。而当与该进程关联的窗口不再拥有焦点时，相应的优先级类别应当恢复为其先前所设置的级别。
- 当一个窗口接收到诸如鼠标事件、定时器事件或者键盘输入之类的输入时，调度器应当提升拥有该窗口的相应线程的动态优先级。



图 18-2 NT 系统中线程优先级关系

在一个线程的动态优先级被提升之后，每当该线程完成一个时间片，调度器都会将该线程的优先级降低一级，直到对应线程降回到其基本优先级别上。也就是说，线程的动态优先级永远不会低于其基本优先级。

NT 系统拥有一些运行的线程被其认为是“实时”性质的线程，其中包括处理如移动鼠标等时间敏感设备之类的线程。所有的从 16~31 之间的优先级被认为是实时的优先级。一个普通的由用户创建的进程所运行的线程往往被赋予从 0~15 之间的优先级。大多数的实时线程是操作系统的线程，但是，用户进程也可以使用实时线程。NT 操作系统不是硬实时系统，所以这些进程都是软实时进程。也就是说，尽管 NT 操作系统会努力确保相关线程根据需要经常运行和尽快运行，但是该系统并不会采取任何措施来保证满足任何即时性标准。另外，NT 操作系统不会提升实时线程的优先级。

当没有其他线程准备好要运行的时候，NT 系统会运行一个称为空闲线程（idle thread）的特殊线程。如果得到了电源电路的支持，这个线程还可使处理器进入运行得较为缓慢的低功耗运行状态，随后，空闲线程便陷入一种紧密型循环之中。拥有这种特殊线程还允许操作系统能够确定究竟有多少系统资源正在被用于实际的工作以及有多少系统资源并没有被使用，后者往往是因为系统正在等待某事发生——可能是等待用户就运行其他哪个程序做出指示。当正在运行诸如 BOINC 之类的志愿者计算软件包时，空闲线程将会被相应的志愿者应用程序所替代。关于志愿者计算项目的相关内容，前面在第 7 章曾经讨论过，在此不再赘述。

424

18.4 内存管理

NT 操作系统支持如第 11 章所述的请求调页（demand paging，或称为请求分页）式虚拟内存（virtual memory）系统。当处理器运行一个进程时，它会产生逻辑内存地址（logical memory address），以方便内存硬件用于提取或存储指令和数据。内存管理部件（Memory Management Unit，MMU）相关硬件则会把所生成的每一个这样的地址转换为存储器系统随后用来访问相应信息的物理地址（physical address）。整个存储器被划分成固定大小的一系

列页面。页面大小由相应硬件决定，因此操作系统必须基于硬件所使用的页面大小来进行工作。对于英特尔 x86 系列来说，有关页面的大小通常为 4KB。但对于其他的系统而言，页面大小可能会有所不同。鉴于 NT 操作系统被设计为是与平台无关的，所以它不必依赖于这些页面的实际大小。同时，考虑到 NT 操作系统被设计成是能够运行在许多硬件平台上的，因此它必须按照灵活适应页面大小的方式来进行编码。

18.4.1 地址空间

在 NT 操作系统中，逻辑地址空间被划分为两部分，一部分用于操作系统，而另一部分则用于用户应用程序，如图 18-3 所示。从图示来看，用户空间和内核空间平均分配了整个地址空间（address space），但是，这在某些情况下有可能被推翻。例如，如果数据库服务器（database server）等应用程序需要一个非常大的内存空间，那么这种均分方案就会被调整。逻辑寻址空间中预留出了两块其他的区域用来辅助检测错误，它们分别被称为警戒区和空指针捕获器。如果一个程序偶然引用了这两片空间中的任何一个地址，那么有关硬件将会发出信号以示错误，并且相应程序将会被中止。（这种机制是由通常与操作系统一起使用的语言支持所采用的一种惯例，而且事实上并不由操作系统本身来强制执行。）别忘了，这一地址是 4GB 的逻辑地址空间的一部分。设立这些保留块的目的在于，当有关地址被意外使用时，通过相关硬件来产生中断。这意味着，并没有任何物理内存会被分配给这些地址，因此我们不会为了这些功能的实现而浪费任何实际的内存。

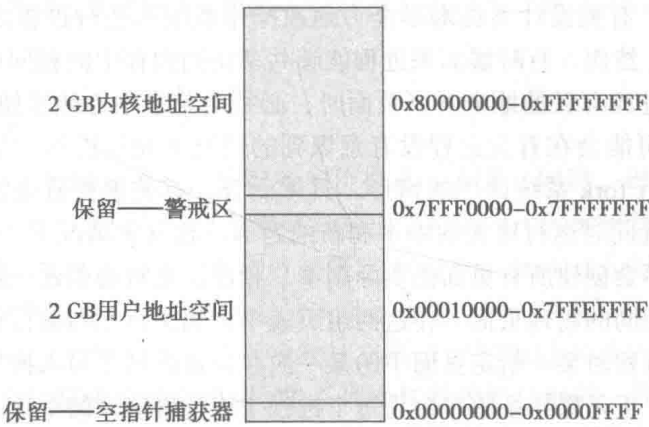


图 18-3 缺省情况下 Windows NT 系列操作系统关于 x86 体系结构的内存映射

425

在与英特尔架构兼容的新型处理器中有一项特殊的硬件特性，支持内核为其自身使用 4MB 大小的页面，从而仅需保存更小的页表（page table），并且有关页表中仅仅需要少数几个表项来指向内核的静态部分以及可能被分页的内核部分的一些其他的页面。

18.4.2 页面映射

在英特尔 8x86 兼容处理器上运行的 NT 操作系统采用两级页表结构和特殊硬件来进行这种页面映射转换，就像第 11 章所描述的那样[⊖]。图 18-4 与第 11 章中的图示相类似，但标以了在 NT 操作系统中所采用的特定术语。其间所用到的这两张表分别称为页目录表（page directory）和页表（page table）。两级页表机制的使用允许逻辑地址空间可以非常庞大，同

⊖ 其他的硬件平台可能会采用更为复杂的设计方案。英特尔兼容的处理器必须是至少 80386 以上的架构。

时也变得非常稀疏，并且页表本身也被划分成为一系列页面。因此，如果在逻辑地址空间的给定块中没有产生页表项，那么就不应创建相应的页表。换句话说，页表中的有关表项应当指向物理内存中的实际页框。

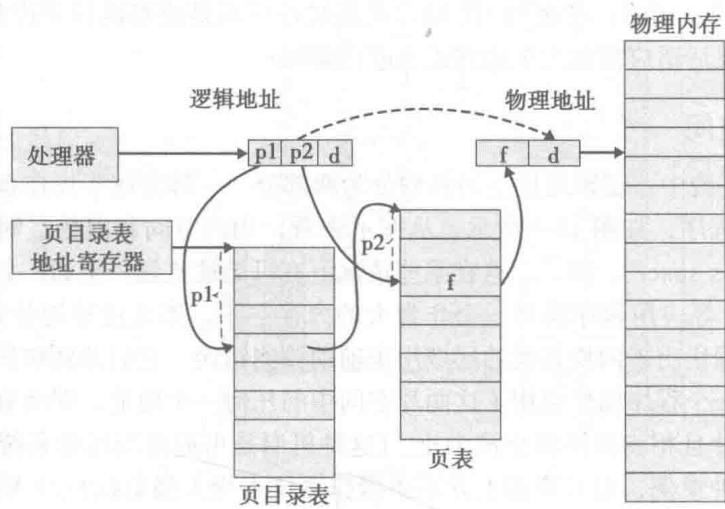


图 18-4 多级页表

18.4.3 分页共享及写时复制

我们可以看到，有关设计人员竭尽全力地在操作系统和运行进程之间以及各进程彼此之间建立隔离机制。然而，有时候如果进程能够共享访问内存中的相同位置，则是非常有利的。当然，在相关进程有目的地来共享页面时，必须要非常小心地来使用此项技术。不过，426有时候操作系统也可能会在有关进程没有意识到的情况下允许进程共享页面。一个简单的例子是一个进程执行 fork 系统调用的情形。具体而言，有关进程请求操作系统创建该进程的另一个副本，并且同时运行原先的副本和新的副本。在这种情况下，NT 操作系统将会创建第二个进程，但不会创建所有页面的实际副本。相反，它将会创建一组新的页表，并将相关页表项指向完全相同的物理页面。在这两组页表中，相关页面均被标记为只读属性。接下来，如果其中一个进程对某一给定页面中的某一内存位置执行了写入操作，那么相应处理器将会产生一个中断，于是操作系统将会为每个进程生成一份该页的单独副本，同时去除只读标志。这项技术称为**写时复制**（copy on write，或称为写时拷贝），曾经在第 11 章中进行过讨论。对于大型对象特别是永远不会发生修改的共享库来说，写时复制技术将会节省大量的时间开销和内存空间。

18.4.4 页面置换

在第 11 章中，我们讨论了当页面需要被加载到内存但却没有空闲页框可用时所出现的问题。在这种情况下，必须得选择一个当前在用的页面来加以置换。我们讨论了许多用来选择欲淘汰页面的算法，其中许多算法利用了硬件功能来辅助操作系统选择要替换的页面。由于 NT 操作系统是被设计为相对独立于硬件的，所以有关设计人员并没有选择依赖于那些供分页硬件所使用的最先进的功能。相反，他们采用了（相对）简单的先进先出置换算法（与时钟式算法存在一定渊源）。进一步说，当一个页面第一次加载进入内存时，应当为对应页面记录一个时间戳。当需要淘汰一个页面时，将会搜索页表，并且选择最先进入内存的页面

并将其替换掉。遗憾的是，相应页面有时会被证实是一个经常需要访问的页面。但是，一旦该页面被重新加载，它将会得到一个新的时间戳，故而，很可能不会马上被再次选中。另外，NT 操作系统关于淘汰页面的选择范围仅仅限于发生缺页的进程自身的页面，即采用局部置换（local replacement）策略。相比之下，Linux 和大多数其他的 UNIX 衍生版操作系统往往采用全局置换（global replacement）策略，即从内存中的所有页面范围内来选择要淘汰的页面。

18.4.5 预取配置

NT 及其他操作系统采用了一种精巧的优化方案来提高应用程序的加载速度。如前所述，当一道程序在虚拟存储系统中启动运行时，整道程序并不是一下子全部加载到内存的。相反，当进程引用了其尚未加载的逻辑地址空间的某一部分时，便会发生一个缺页故障，然后将所需要的页面加载到物理内存中。为此，当加载一个大的程序（譬如网络浏览器）时，伴随用于各种数据结构的初始化代码的运行，该程序的不同部分的页面更倾向于几乎是以随机顺序加载到内存的。于是，当提取各段程序代码时，会发生相当多的磁盘操作和磁头移动。通常情况下，NT 操作系统会监控一道程序启动执行 10 秒内所产生的缺页故障。然后，在系统不忙的时候，它就会对这份缺页故障列表进行排序，并将相关结果列表保存在该程序的缺页故障配置文件（page fault profile）中。当以后再次运行相应程序时，操作系统将会预先读取其所掌握的通常在前 10 秒内应当加载的所有页面。这种方案可以极大地减少磁头的移动操作以及旋转延迟和较大盘块输入/输出等待所带来的时间开销，并因此会导致更快捷的程序启动速度。

427

18.5 文件支持

在过去 20 年左右的时间里，计算设备的价格迅速下降。同时，磁盘存储器的容量却迅猛增加。当磁盘容量很小的时候，文件系统结构应当按照与小容量磁盘相匹配的方案来加以设计。为此，早期的 DOS 文件系统的指针被限定为 12 位，因为这足以用来指向当时所使用的软盘驱动器上的任何扇区，并且有关设计人员不想把空间“浪费”在“较大”的指针上面。然而，伴随驱动器容量的增大，文件系统设计必须进行调整以支持更大的硬盘驱动器。同时，作为将现有个人计算机系统升级到 NT 系列操作系统的目标的一部分，也需要为用户可能拥有的各种各样的文件系统建立一条迁移途径。毋庸置疑，大多数的操作系统都有自己首选的文件系统。NT 操作系统也不例外，相应的文件系统被称为 NTFS（NT File System，或称为 NT 文件系统）。但是，NT 操作系统也支持其他的文件系统，特别是微软早期所开发的那些文件系统，包括从 DOS 和 Windows 系统那里继承来的 FAT12、FAT16 和 FAT32 文件系统。XP 操作系统还支持面向光盘（CD）的 ISO 9660 只读光盘标准格式文件系统（即 NT 系统中的 CDFS）、UDF（Universal Disc Format，即统一光盘格式）文件系统、面向可擦写的光盘和数字化视频光盘的 ISO 13346 标准格式文件系统，源自 OS/2 操作系统的 HPFS（High Performance File System，即高性能文件系统）以及许许多多其他的标准文件系统。考虑到没有更换掉 OS/2 操作系统的机器数量太少，故而 NT 操作系统后来无暇顾及并最终抛弃了对 HPFS 的支持。在开发 NT 操作系统的时候，在 80x86 机器上尚没有大量地安装任何一版的 UNIX 操作系统，于是微软认为显然没有必要去支持任何特定的 UNIX 文件系统，所以就没有在这方面付诸任何努力。

18.5.1 NTFS

我们首先介绍 NTFS 的几项一般特性，然后讨论 NTFS 的主要目标以及这些目标是如何实现的。最后，我们将就 NTFS 的一些高级功能特征展开进一步的论述。NTFS 卷的基本布局如图 18-5 所示。

主文件表

NTFS 卷的引导扇区包含有一个指针指向主文件表（Master File Table，MFT）。文件系统必须记录大量的关于文件的元数据（与文件自身所包含的数据相对而言）。关于 NTFS 卷自身的关键元数据则作为特殊的系统文件而存储在主文件表中。NTFS 卷中的每个文件或目录在主文件表中都有一条对应的 1024~4096 字节长的记录^①。关于文件和目录的元数据作为属性（attribute）存放在主文件表的记录中。属性就是我们通常所认为的文件系统目录项的相关字段。鉴于给定文件所需的属性往往可能因文件的类型而有很大变化，所以大多数的属性被存储为键值对（pair），即一个标识符和一个取值。其中某些属性总会出现，故而存放在相应文件的主文件表表项的前面，但大多数的属性都位于可变的序列中。鉴于每条主文件表记录的大小是有限长的，所以 NTFS 采取了两种不同的方式来存放文件的有关属性：或者作为常驻属性（resident attribute）存放在主文件表记录（MFT record）中，或者作为非常驻属性（nonresident attribute）存放在其他的主文件表记录中或文件系统的非主文件表簇中的扩展部分。

428

- 常驻属性。仅仅要求少量空间的属性往往保存在相应文件的主文件表记录中。大多数常见的文件属性都是常驻性质的。其中，一些属性要求是常驻性的，例如，包括文件名以及文件创建、修改和访问的日期（或称为时间戳）总是常驻性的。图 18-6 显示了一条具有常驻属性的主文件表记录。

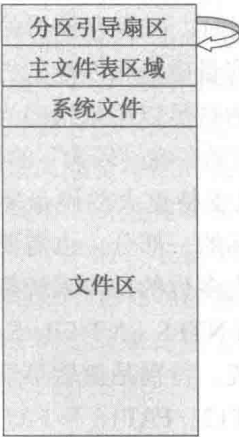


图 18-5 NTFS 卷布局



图 18-6 具有常驻属性的主文件表记录

- 非常驻属性（亦称外部属性，即 external attribute）。如果一项属性无法存放在主文件表记录中，就把它放在一个单独的地方，同时通过在主文件表中设立一个指针给出相应属性所存放的位置。非常驻属性的存储分为两种情况。首先，如果指向属性取值的指针可以存放在相应文件的主文件表记录中，那么相应的取值就被存放在主文件表记录之外的称为扩展盘块区（extent）的数据盘区（data run，或称为数据行程）

① 从技术上来讲，NTFS 是专利性质的，所以一些细节往往是通过观察推断出来的，故而经常会存在争议。

中，并且指向相应数据盘区的指针被放置在主文件表记录中。（这对于数据属性而言在大多数情况下都是成立的，但从理论上讲，这可以应用于任何属性。）图 18-7 显示了一条具有非常驻属性的主文件表记录。其次，一项属性也可能存放在许多不同的盘区（run，或称为行程）中，且每个盘区具有一个单独的指针。如果有关属性取值具有非常多的扩展盘块区，以至于指向相关扩展盘块区的指针甚至都无法存放在主文件表记录中，故而整个属性就可能要外迁成为一条单独的主文件表记录的外部属性甚至是多条外部记录。

429

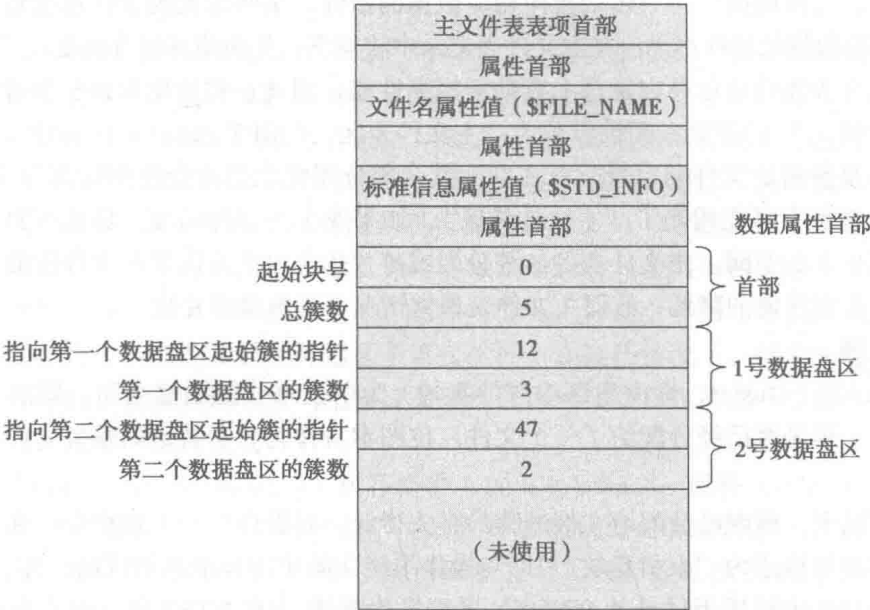


图 18-7 具有非常驻数据属性的主文件表记录

NTFS 拥有若干预定义的属性。其中，一些属性仅仅与一个文件或一个目录相关联，或者仅仅与有关卷的元数据中的某种其他结构相关联，而另外的属性则与一个以上的结构相关联。这里是一些最常见的 NTFS 系统所定义的属性：

- 卷名 (volume name)、卷信息 (volume information) 和卷版本 (volume version)。指卷自身的键名、版本和其他的元数据。
- 位图。指包含有关于簇的分配信息的位图。这一属性仅会被有关位图元数据的主文件表记录所使用。
- 文件名。指文件或目录的名称。一个文件或目录可能具有多个文件名属性，以允许微软 DOS 操作系统的短文件名或者支持 POSIX 实现对源自多个目录的硬链接。
- 标准信息。指所有的文件和目录都需要的数据，譬如文件创建、修改和访问的日期或时间戳，以及只读、隐藏等。
- 索引根 (index root)。指针对一个目录中的所有文件的索引。如果目录很小，那么整个索引都可以存放在主文件表中。否则，某些信息将会被作为非常驻属性加以处理。
- 安全描述符 (security descriptor)。指控制对文件或目录的访问的信息（例如，所有权、访问控制表和审计信息）。
- 属性列表 (attribute list)。这是一个元属性 (meta-attribute)，它用来描述其他的属性。如果一项属性是非常驻属性，那么该属性的标识符将被放置在主文件表记录中，同

430

时拥有一个指针指向相应的非常驻属性。

- 数据。一个文件中的数据就是“数据”(data)属性的取值。如果一个文件的所有属性(包括数据)都能够在主文件表记录中放得下,那么相应数据属性将会驻留在主文件表记录中。这样的文件不需要卷上的其他存储空间,更为重要的是,它们不需要额外的磁盘访问来读取数据,从而有利于性能的提高。

越大的文件越复杂。如果一个文件的所有属性均无法在主文件表记录中放得下,那么有关属性将被设置为非常驻性质。因此,大多数的文件往往把它们的数据存放在主文件表记录之外。显然,文件的属性应当包括指向相应数据的指针。某些很大的文件可能特别大,以至于连指向相应数据的属性都无法在主文件表记录中放得下,从而使其自身也变成了外部属性。

把主文件表连续地保存在磁盘上有助于提高性能,因此在初始化 NTFS 卷时,紧跟主文件表之后大约 13% 的磁盘空间被保留为“主文件表区”(MFT zone)。这部分空间也是可以利用的,但是普通的文件和目录只有在空间其余部分用完之后才会使用这部分空间。同时,主文件表最终也有可能用完了“主文件表区”。如果发生了这种情况,那么 NTFS 将会为主文件表分配更多的空间。主文件表的这种分裂或碎片化有可能会因某些文件所需读取操作数量的增加而造成性能的降低,故而主文件表通常情况下不应被碎片化。

空间监测

NTFS 以簇(cluster,即由扇区组成的盘块)为单位来分配磁盘空间。同时,其采用位图来监测每个簇是否已经分配给了一个文件。位图本身作为一个特殊的系统文件而存放在主文件表中。

通常情况下,指向已分配给文件的簇的有关指针一起保存在一个盘块中。在第 12 章中,我们把这些盘块描述为“索引盘块”,以与操作系统文献中的标准术语保持一致。(这一术语不应当与 NTFS 中适用于目录的 \$INDEX 属性发生混淆。)在 NTFS 中,有关索引指针给出一个数据盘区(data run,或称为数据行程)的起始的簇号。其中,数据盘区是指一个连续的簇群,它们均被分配给了同一文件。除了拥有一个起始簇的编号之外,相应数据盘区的长度(即数据盘区中所包含的连续的簇的计数)也被记录在对应的数据盘区表项中。采用这种数据盘区的主文件表记录如图 18-7 所示。

NTFS 主要目标

NTFS 拥有两项主要目标:高可靠性(high reliability)和安全性(security)。高可靠性是通过两个不同的方向,即崩溃后的可恢复性以及软件数据冗余和容错(主要是廉价磁盘冗余技术)同时努力来达成的。当然,除了这三项主要目标之外,NTFS 还提供了许多其他的高级功能特征。

1) 可恢复性(recoverability)。或许,NTFS 设计的主要目标是增加文件系统在面对崩溃时的可靠性。对于以往的文件系统设计方案而言,如果控制整个文件系统的数据由于系统异常关机而遭到破坏,那么所有的文件或者大部分文件极有可能就被丢失和无法恢复了。为了增强文件系统的可恢复性而演化形成的机制就是在第 13 章所讨论的基于日志的文件系统(log-based file system)或日志式文件系统(Journaling File System, JFS)。每当对文件系统的元数据实施任何的更新操作时,NT 系统首先会写一条记录到日志文件中,以罗列所需完成的更新操作步骤,相关这组步骤被称为事务(transaction)。然后,再逐一实施每一步更新操作。最后,还需就元数据更新所涉及的相关所有文件执行相应的输入/输出操作。一旦这一系列步骤全部完成,那么关于相关步骤的记录应当从日志文件中删除。如果系统停止了

运转,那么当其重新启动和恢复时,应检查确认是否存在某项更新事务正在进行中。如果日志文件中的某条记录表明正在进行更新操作,那么操作系统应当能够识别出哪部分操作尚未成功完成。进一步说,如果系统可以完成该项事务,就继续实施完成该项事务;而如果系统不能完成有关事务,则需要就已经完成的部分实施恢复操作。这样,整个文件系统将始终处于有效状态。即便某些应用程序的数据有可能已被丢失,但至少文件系统可以继续使用,同时无须担忧将来还有其他数据会因文件系统已陷入损坏状态而被丢失。Vista 版操作系统还包括所谓卷影复制和事务支持的功能选项,可以为数据文件提供保护。相关特性将会在第 18.5.5 节中展开进一步讨论。

当然,这些额外的步骤会需要额外的时间开销,并会增加磁盘驱动器的负载。不过,鉴于 NT 操作系统主要是为个人计算机而设计的,所以额外的负载还是可以容忍的。毕竟,系统在大多数情况下并不可能是过载工作的。同时,NTFS 的其他设计要素也使得其某些性能增益要高于 NT 系统所支持的其他文件系统。因此,鉴于 NTFS 相比于其他微软文件系统所具有的更高的可靠性,NTFS 的整体性能还是可以接受的。另一方面,如果系统发生崩溃且有关文件系统不是基于日志的系统,那么保险的做法是运行一个实用例程来检查文件系统的完整性。但在拥有许许多多文件的系统上,这将可能花上好几个小时才可能完成。显然,在用作服务器的系统上,如此漫长的延迟是不可接受的。在这些情况下,把有关性能影响分散开来是更为合理的,也就是说,应当把相应完整性的保证分布在正常的日常操作中,而不是在系统崩溃后触发和一次性完成。

2) 数据冗余 (data redundancy) 和容错性 (fault tolerance, 或称为容错能力)。NT 系统的另一项磁盘可靠性特征是支持三种不同的软件版廉价磁盘冗余阵列 (Redundant Array of Independent Disk, RAID) 构型。关于廉价磁盘冗余阵列,曾经在第 14 章进行过较为详尽的讨论。NT 操作系统支持的廉价磁盘冗余阵列形式包括 RAID-0、RAID-1 和 RAID-5。RAID-0 完完全全是用来增强性能的,没有提供更高的可靠性。RAID-1 则是完全镜像,也就是说,写入一个驱动器的所有内容都会自动写到另一个驱动器,故而提供了良好的可靠性,不过需要付出较高的硬件成本。对于 RAID-5 来说,对应于一组数据块会同时写入一个奇偶校验块,其以较低的硬件成本提供了良好的可靠性,但会增加软件开销。当然,硬件版廉价磁盘冗余阵列系统可以与 NT 操作系统联合使用,而不是局限于使用软件解决方案层面上。

3) 安全性。在 NT 系统中,对象是操作系统所有数据结构的基本构件,在这些数据结构中就包括文件和目录。每一个对象都有一个所有者,即最初创建相应对象的实体。利用访问控制表 (Access Control List, ACL), 安全性可以施加到任何对象上。或许你还记得,一个实体的访问控制表列出了允许操作对象的有关实体 (包括组和角色) 以及允许对应实体所执行的相应操作列表。所有者可以对访问控制表执行若干种操作,包括直接对其进行修改、授权其他实体对其进行修改以及设置其他实体成为所有者。在 NTFS 中,文件或目录的访问控制表存储为相应对象的属性。NTFS 中所使用的权限如下:

- * R——读
- * W——写
- * X——执行
- * D——删除
- * P——修改访问控制表
- * O——将当前账户设为新的所有者 (“获得所有权”)

18.5.2 NTFS 高级功能特征

NTFS 包含有许多支持应用程序的高级功能。其中一些功能以应用程序接口调用的方式供应用程序使用，而另外一些功能则只能在其内部使用：

- 只读支持 (read-only support)。在 XP 版操作系统之前的 NTFS 要求卷位于可写介质上面，以便其能够写操作事务日志文件。XP 操作系统则引入了可以在只读介质上挂载卷的驱动程序。对于拥有 NTFS 格式只读卷的嵌入式系统来说，这项功能非常必要。
- 碎片整理 (defragmentation)。NTFS 并没有特别设法来保持文件存储的连续性。但是，它提供了一个关于碎片整理的应用程序接口，相关应用程序可以通过调用该接口来迁移文件数据，从而使有关文件占用连续的簇。NT 操作系统还包含有一个碎片整理工具，不过该工具存在若干方面的限制。相比之下，第三方提供的此类产品往往拥有更为丰富的功能。
- 卷挂载点 (volume mount point)。这类似于 UNIX 系统的挂载点。在 NTFS 中，这便支持其他文件系统的可见性且无须为每个文件系统配备单独的驱动器号。对于远程卷也是如此。
- POSIX 支持。NT 操作系统的目标之一是支持 POSIX 标准。对于文件系统来说，这便要求支持区分字母大小写的文件名和目录名、一种不同的在解析路径名时确定访问权限的方法以及不同的时间戳语义集。这些功能与 NT 系统自身均不兼容。NTFS 包含有这些可选的支持 POSIX 的功能。
- 加密。当笔记本电脑丢失或被盗时，存储在笔记本电脑上的数据有可能被暴露。在这种情况下，文件系统保护做得并不到位，因为相关卷可以由不要求 NT 操作系统运行的软件来读取。此外，当另一个用户可以使用具有管理员特权的账户时，NTFS 的文件权限是毫无用处的。因此 NTFS 包含了一项称为加密文件系统 (Encrypting File System, EFS) 的功能来加密存储在数据属性中的数据。加密文件系统对应用程序是完全透明的。只有利用账户的加密文件系统的私钥 / 公钥对中的私钥才能访问加密文件，并且私钥被采用账户的密码进行了锁定，所以在没有授权账户的密码的情况下是无法读取对应的文件的。
- 卷影复制 (volume shadow copy)。这项服务通过把被覆盖 (即改写前) 的数据复制到隐藏的影子备份区域，从而保存了 NTFS 卷上的文件和文件夹的历史版本。于是，有关用户以后就能够请求切换回较早的版本。这项功能允许备份程序归档当前正在使用的文件。
- 链接跟踪 (link tracking)。快捷方式允许用户在其桌面上放置文件。类似地，对象链接和嵌入 (Object Linking and Embedding, OLE) 允许源自一个应用程序的文档被链接为其他应用程序的文档。这种链接提供了一种简单的使文件彼此连接的方法，但是这些链接比较难以管理，因为如果有关用户移动了一个链接对应的目标，那么相应链接将会被断开。为此，NTFS 支持分布式链接跟踪，也就是说，当链接目标发生移动时，系统将会维护源自命令解释器命令的文件链接以及对象链接和嵌入中相关链接的完整性。利用 NTFS 链接跟踪支持，如果一个位于 NTFS 卷上的链接目标移动到了相同域中的另一个 NTFS 卷上，那么有关链接跟踪服务能够同步更新链接以反映这种移动情况。

- 单实例存储 (Single Instance Storage, SIS)。有时, 几个目录均包含具有相同内容的文件。单实例存储允许把相同的文件缩减为一个物理文件以及对该物理文件的许多单实例存储引用。单实例存储是一个用于管理文件变化的文件系统过滤器以及一项用于搜索内容相同且需要合并的文件的服务。与仅仅指向一个文件的硬链接不同, 在文件系统的外部来看, 每个单实例存储文件仍然是不同的, 并且对一个文件的一个副本的改变不会改变其他的文件。对于发生更改的一个单实例存储文件来说, 系统将会为其另外创建一个不同的副本。
- 每用户磁盘空间配额 (per-user disk space quota)。管理员经常需要监测或限制用户对磁盘空间的使用情况, 特别是在服务器上, 因此 NTFS 包含了对配额管理的支持, 允许为每个用户指定相应的磁盘空间配额。
- 更改日志 (change logging)。有时, 应用程序需要监控卷上的文件和目录的更改情况。例如, 当文件发生变化时, 自动备份程序可能会执行增量备份 (incremental backup)。为了实现这一目标, 一种方案是相关应用程序扫描对应卷并记录文件和目录的状态, 然后在稍后的扫描过程中, 进一步检查前后是否存在差异。然而, 有关过程显然可能会使系统缓慢下来, 尤其是在计算机通常拥有成千上万的文件的情况下。为此, NTFS 允许应用程序请求 NTFS 把有关文件和目录更改的信息记录到一个称为变更日志 (change journal) 的特殊文件中。在此基础之上, 相关应用程序可以读取更改日志, 而不是去扫描整个目录树。
- 事务支持。使用 Vista 操作系统, 应用程序可以采用事务把文件的更改操作划分和编组到一个事务中。事务保证要么所有的更改都发生, 要么任何更改都不发生, 并且将会保证事务外的应用程序只有在事务提交之后才会看到对应的更改结果。事务通常在数据库系统和 NTFS 元数据中都得到了支持。这项功能把基于事务的系统的可靠性延伸到了普通文件上。
- 压缩 (compression) 和稀疏文件 (sparse file)。NTFS 支持压缩文件数据。压缩和解压缩 (decompression) 是透明的, 因此不必修改应用程序就能够利用这项功能。目录也可以被压缩, 同时被压缩目录中的任何文件都会自动地被压缩。NTFS 还有一种称为稀疏文件的相关机制。进一步说, 如果一个文件被标记为稀疏特性, 那么 NTFS 将不会在卷上为该文件的空白部分 (即无效部分) 分配空间。当应用程序从稀疏文件的空白区域读取数据时, NTFS 将会返回用 0 填充的缓冲区。与压缩文件 (compressed file) 一样, 稀疏文件通常对于应用程序也是透明的, 尽管当处理文件中实际上为空的的部分的时候, 应用程序可能会意识到稀疏文件并且可能会节省大量的处理器和内存资源开销。
- 别名 (alias)。NTFS 支持硬链接 (hard link) 和符号链接 (symbolic link)。硬链接允许多个路径引用同一个文件, 相关实现与第 12 章中讨论的一样。为了防止陷入查找循环, NTFS 采用了一种简单的方法, 即不允许硬链接对目录进行引用。另外, NTFS 把符号链接称为**联结点** (junction), 而联结点则建立在一种所谓**重解析点** (reparse point) 的更通用的机制的基础之上。重解析点是指能够被输入/输出管理器所读取的关于文件或目录的额外属性, 例如相应文件或目录的当前位置。在文件或目录查找期间, 当 NTFS 命中一个重解析点时, 它将会告知输入/输出管理器检查相应的重解析数据 (reparse data)。然后, 输入/输出管理器可以更改原先操作中所指

定的路径名，并让 NTFS 用更改后的路径重新开始查找。重解析点也可以被磁带归档软件所使用以说明一个文件已被迁移到归档系统。有关磁带归档软件把文件移动到磁带上，并在其目录项中留下重解析点，以告知相关软件对应文件当前所在的位置。当进程尝试访问一个已经归档的文件时，驱动程序将会从对应目录中移除重解析点属性，把文件数据从归档介质读回到原先的介质上，然后重新进行访问。因此，对于访问归档文件的进程来说，离线数据的检索是透明的。当然，在这种情况下的文件打开操作可能需要花费比正常情况稍长一点的时间。

- 动态坏簇处理 (dynamic bad-cluster handling)。如果数据读取操作访问了一个损坏的磁盘扇区，那么读操作失败，并且有关数据无法再用。然而，如果对应磁盘是具有容错能力 (譬如廉价磁盘冗余阵列) 的卷，则驱动器将会获取相应数据的一份好的副本，并且告知 NTFS 对应扇区已经损坏。于是，NTFS 将会在发生故障的驱动器上分配一个新簇来替代坏簇，并把有关数据复制到那里。鉴于 NTFS 已经对坏簇进行了标记，故而以后将会对其忽略和不予理睬。
- 索引 (indexing)。NTFS 支持对磁盘卷上的任何文件属性进行索引。索引会对有关属性进行排序。利用索引机制，可使文件系统能够快速地找到满足任何条件的文件，例如去找出某个目录中的所有文件。
- 复杂文件名 (complex file name)。NTFS 使用 Unicode 字符来存储文件、目录和卷的名称。Unicode 是一种 16 位字符编码方案，支持世界上的每种主要语言中的每个字符被唯一地表示。路径名称中的每项要素最多可以包含 255 个字符，并且可以包含 Unicode 字符、空格和多个句点。
- 多个数据流 (multiple data stream，或称之为多重数据流)。在 NTFS 中，文件的数据被认为是对应文件的一种属性，称为数据流 (data stream)。可以由应用程序添加新的属性，包括附加的数据流，所以文件 (和目录) 可能包含多个数据流。NT 操作系统利用一个备用的数据流 (简称备用流) 把用户的“特性”与文件关联到一起，例如标题、主题、作者和关键字。同时，它将日期存储在称为摘要信息 (summary information) 的备用流中。

435

18.6 基本输入 / 输出

从图 18-8 中，我们可以更为清晰地看到 NT 操作系统的整个文件系统的体系结构。遗憾的是，就像操作系统文献经常发生的那样，我们在本教材中所使用的名称与 NT 系统设计人员所使用的名称往往会发生冲突。例如，他们把输入 / 输出系统的顶层称为“输入 / 输出管理器”，而我们则已经将该术语用于操作系统的较低层级的输入 / 输出功能上。在本章中，我们将采用微软所使用的术语。因此，我们在上一节中所描述的功能实际上是由分区 / 卷存储管理器和磁盘类管理器所承载的。

18.6.1 分区

因为个人计算机中的硬盘驱动器支持是源自于微软或 DOS 操作系统所采用的设计方案，所以个人计算机上的任何操作系统都会支持某些相同的事情。首先，有关设计支持系统管理员把磁盘驱动器划分为所谓分区 (partition) 的单独区域。在此基础上，管理员将会使用操作系统实用程序在每个分区中建立一个单独的文件系统。这些文件系统甚至可以是其他操作

系统原生的文件系统。输入 / 输出系统把每个分区看作是一个单独的驱动器。与文件系统一样，分区结构的设计不得不演化改进以应对硬盘的不断增大的容量。分区机制在硬盘的所谓主引导块（Master Boot Block，MBB）或主引导记录（Master Boot Record，MBR）的第一扇区中创建了一张小表。原有机制在单个磁盘上只能创建 4 个分区。后来进行了扩展，允许一个分区被指定为扩展分区（extended partition），从而允许在一个磁盘上最多可以创建 24 个逻辑分区。

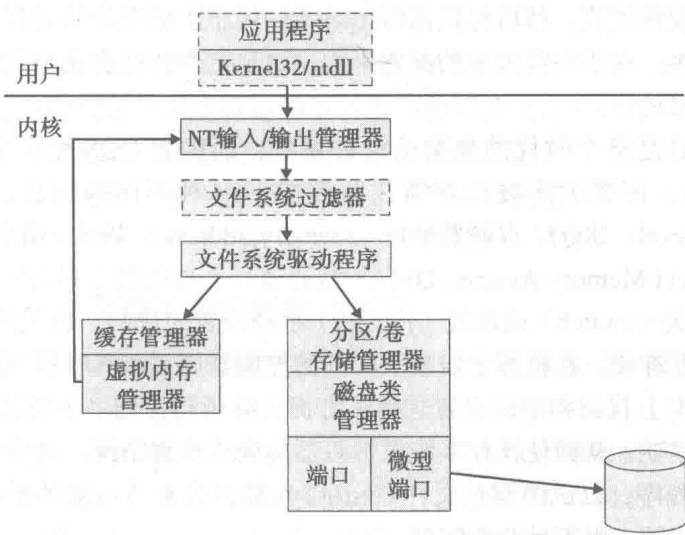


图 18-8 NT 操作系统输入 / 输出体系结构

436

18.6.2 输入 / 输出系统分层

输入 / 输出系统的分层允许把新增的额外层次方便地插入整个操作系统的体系结构中。在许多情况下，NT 输入 / 输出驱动程序会在它们的供其用来调用下一较低层级驱动程序的应用程序接口上公开相同的调用，并且任何给定层级的每个替代模块都会实现相同的接口。除此之外，这种分层还允许把一个逻辑设备定义在一个实际上并非本地磁盘分区而是位于网络上的另一台机器的系统上。在这种情况下，系统将会执行一个重定向（redirection）操作，从而使得对于用户和程序来说，网络设备看起来与本地设备并没有什么区别。有关分层还允许某个设备看起来就像是磁盘驱动器，但它实际上是别的什么东西——譬如一个通用串行总线接口标准的闪存驱动器。

这种分层还允许在层级之间插入额外的功能。它们将会在系统引导时以一种所谓过滤器（filter）的特殊类型的设备驱动程序的形式进行加载。对于一般普通用户可能并不想要任何额外功能的简单情况来说，仅需付出很少的开销便可以支持基本的输入 / 输出功能。但当一个用户确实想要某种外来的功能时，有关补充的功能可以采取一种无论对较高层级还是对较低层级都透明的方式而插入两个层级之间。病毒扫描器就是此类补充功能的一个例子。通过在层级之间提供有关接口，NT 操作系统就能够支持第三方软件方便地扩展输入 / 输出系统的功能，且不会破坏操作系统代码的完整性。还有，由于这种定义良好的标准分层接口，如果有任何意料之外的功能将来被开发出来，也很容易就能够被添加到输入 / 输出系统中。

18.6.3 即插即用

在编写操作系统时，其被认为是一个通用实体，能够运行在各种各样的硬件配置平台

上。但是，当我们在一台特定的机器上安装操作系统的时候，必须按照与所安装的硬件相匹配的方式来对其进行配置。如果以后添加了新硬件或者更换或移除了旧硬件，则必须对操作系统进行调整以适应新的机器配置。我们需要新硬件的驱动程序，同时我们也不想把空间浪费给那些不再需要的驱动程序。

对于早期的大型主机来说，在安装操作系统或对操作系统进行升级的时候，常常会看到系统编程人员要执行一条 `sysgen` 命令（用于生成系统）。简单来说，这就相当于利用一个规格记录文件来描述硬件配置，然后将该规格记录文件用于生成专为适应有关硬件而定制的操作系统的可执行版本。对于中等大小的配置来说，这项工作往往会花上几天，有时需要尝试多次才能够实现正常运行。

原来的 IBM PC 是那个时代的典型的硬件系统，而配置 DOS 操作系统以适应硬件则是非常困难的事情。配置大多数控制器往往需要 2~4 种不同的信息，包括中断请求级（Interrupt Request Level, IRQ）、存储器地址（memory address）、输入/输出端口（地址）以及内存直接存取（Direct Memory Access, DMA）型通道号。这些都是利用控制器板（controller board）上的小型开关（switch）或跳线（jumper）手动设置完成的。有关地址必须谨慎选择，以确保它们不会相互冲突。在机器中安装一个新的控制器则可能具有相当的挑战性，因为通常很难在现有的板卡上找到相应的设置。另一方面，必须得使用一个所谓 `config.sys` 的文件来对相关硬件加以描述，从而使操作系统掌握机器的硬件配置情况。通常情况下，硬件供应商会提供一个实用程序，以试图调整文件 `config.sys` 的内容来适应新的硬件，但是他们往往会由此导致更多的问题，而不是修复问题。

从 IBM MicroChannel（微通道）和 EISA（Extended Industry Standard Architecture，扩展版企业标准架构）总线开始，相关控制器已能够向操作系统标识自己的身份，并能够对软件引发的配置更改做出响应。这项功能被称为即插即用（plug and play，有时简记作 PnP）。使用 PCI 总线后，这种趋势得以延续，并且当前大多数操作系统都能够识别大多数新的硬件、动态地设置板卡的参数、选择能够与现有硬件配置一起正常工作的配置方案，以及通过为有关硬件动态地加载正确的设备驱动程序来定制操作系统。当然操作系统仍然需要调整才能够适应硬件，不过相关过程通常由操作系统动态地来完成，并且对用户来说，有关操作变得更加透明。

18.6.4 设备驱动程序

输入/输出设备的所有硬件特性都被隔离在内核的最低层级，即设备驱动程序中。例如，这便意味着，所有较高层级的模块不应当去关心一条磁道上有多少个扇区或一个磁盘驱动器拥有多少个读/写磁头。同时，它们也不应该关心状态寄存器中的哪些位用来指示发生了错误。相反，它们应该专注于对所有磁盘驱动器来说都通用的东西，并把任何特定设备（或控制器）的细节信息限制在相应特定设备或控制器的设备驱动程序内部。

由于 NT 操作系统使用这样的设备驱动程序来隐藏相关硬件的细节，所以很容易就能够改变 NT 系统的硬件配置。实际上，有关驱动程序可以动态地安装到系统中或者从系统中移除。这意味着，当一部设备添加到系统中时，无须重新启动操作系统。在这种新方案之前，当相关硬件发生改变时，系统重新启动则是必须的。显然，重启系统是非常耗时的，而且对于诸如服务器等非常重要的系统来说，这也是令人难以接受的。对于以物理方式插入总线的设备控制器（例如，新的图形卡）来说，系统电源无论如何必须得关闭，因此，这

种情况下的非得重新启动系统并不是什么问题。然而, 几种用来把外围设备连接到计算机的新方法都假定相应设备是外部的, 比如录像机 (Video Cassette Recorder, VCR) 或便携式摄像机 (camcorder), 因此关闭电源就不再是必要的。此类接口的示例包括通用串行总线 (Universal Serial Bus, USB)、IEEE 1394 以及 PC 卡 (PC Card) 或插件总线 (Card Bus, 以前称为 PCMCIA, 全称是 Personal Computer Memory Card International Association, 即个人计算机内存卡国际联合会) 等。此外, 还为这些接口定义了协议, 使得有关设备可以采用类似于 PCI 总线的即插即用特征的方式向计算机标明其自己的身份。这种动态识别意味着, 对于任何新安装的设备来说, 操作系统可以自动加载相应的驱动程序, 且无须重新启动操作系统; 除了可能需要用户提供包含对应设备的驱动程序的只读光盘外, 通常情况下并不需要用户的任何其他帮助。大多数用户还会在不关闭电源的情况下把设备连接到串行端口或并行端口上, 尽管此类设备的制造商通常不推荐这样做。不过, 通过这些端口连接的设备可能无法自动地标明自己的身份, 这和那些使用较新接口的设备是不一样的。

438

18.6.5 磁盘类、端口及微型端口的驱动程序

一般来说, 文件系统模块会调用位于较低的逐渐向硬件靠拢的层级上的存储驱动程序 (storage driver) 的功能。相关层级分别被称为存储类 (storage class, 这里即磁盘类, 也就是 disk class) 驱动程序、存储端口 (storage port) 驱动程序以及微型端口 (miniport) 驱动程序。在最上层, NT 操作系统提供了存储类驱动程序, 其实现了面向所有特定类型的存储设备 (譬如磁盘或磁带) 的通用功能。在其下一层是存储端口驱动程序, 其承担了诸如小型计算机系统接口 (Small Computer System Interface, SCSI) 或高技术连接系统接口 (Advanced Technology Attachment, ATA, 或称为高技术附加装置) 等特定总线的通用功能。磁盘驱动器供应商往往会提供微型端口驱动程序, 以支持特定设备或系列兼容设备。存储类驱动程序具有与设备驱动程序接口相同的应用程序接口。而微型端口驱动程序使用端口驱动程序接口, 而不是设备驱动程序接口。这种方法简化了微型端口开发人员的工作, 因为它们具有与以往微软操作系统相兼容的应用程序接口。存储类驱动程序通常可以管理和操纵相应存储类中的许多设备, 而无须存储端口驱动程序或微型端口驱动程序。最典型的例子就是一般的通用串行总线存储类驱动程序, 该驱动程序可以在没有任何其他驱动程序的前提下访问许多通用串行总线存储设备。

18.7 图形化用户接口编程

诚然, 对于用户来说, Windows 操作系统的定义的特性就是图形化用户界面。程序设计人员通过 Windows 操作系统的应用程序接口来访问那些操纵桌面上相关对象的操作系统函数。有关接口为编程人员提供了绘制窗口以及制作菜单和对话框等一系列功能。操作系统本身负责通用功能, 例如确保当一个窗口关闭时, 该被关闭窗口后面的相应窗口的适当部分被刷新。同时, 操作系统还为程序设计人员提供了通用对话框等设施。如图 18-9 所示, 该图显示了一个标准的基于窗口的对话框, 而有关程序可以使用该对话框来查找要打开的一个文件 (或多个文件)、指定要保存的文件名称、选择字体或颜色以及任何程序可能需要支持的其他常见功能。应用程序编程人员可以使用这一界面, 但并不是必须要这样做才行。如果使用这一界面, 那么不同的应用程序均会呈现出相似的外观和感觉。当程序设计人员开发完成了更为复杂精致的界面时, 他们往往倾向于使用那些复杂精致的界面来代替这种标准界

439

面。有人可能认为，这些新的界面对用户更加友好或者更适合于给定的任务。但是，如果每个应用程序都有一套不同的界面，那么就有可能使得整个系统对于初学者来说更加难以掌握。因此，尚无法确定，这种权衡是否真的总有必要的。

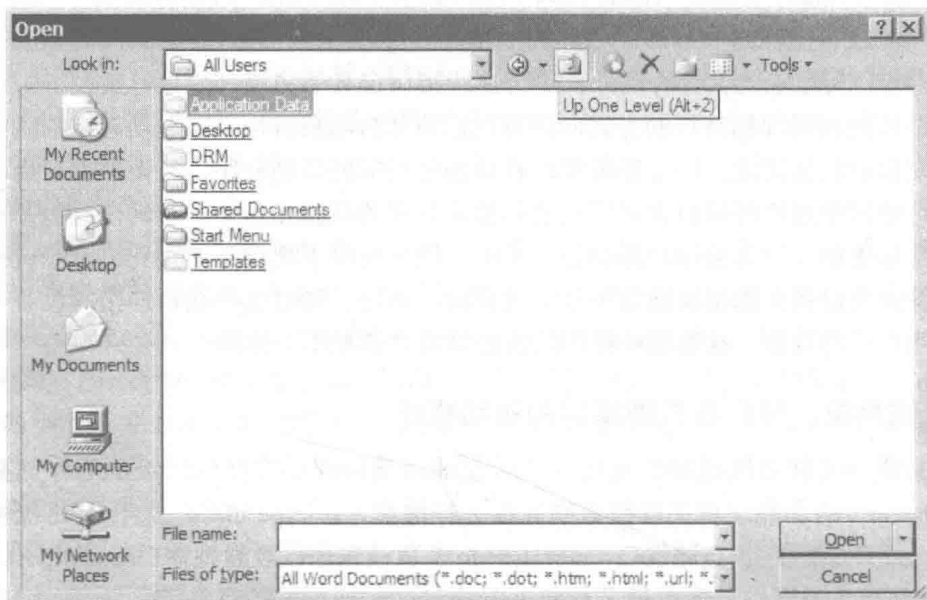


图 18-9 “文件打开”对话框

18.8 网络

有关设计开发人员希望 NT 操作系统能与其他操作系统保持兼容的另一方面是其所支持的网络协议。在 NT 操作系统开发时，互联网在学术界已经相当流行，但是在互联网中所使用的传输控制协议和网际协议还没有成为像今天这样的占主导地位的网络协议。当时，Novell Netware 操作系统是主要的个人计算机文件服务器平台，而且其拥有自己的协议，即互联网数据包交换（Internet Packet Exchange, IPX）协议和序列数据包交换（Sequenced Packet Exchange, SPX）协议。与此同时，还有许多 UNIX 系统在运转着。另外，许多较大型的企业则拥有 IBM 公司的大型主机（mainframe）和中端系统（midrange system），而这些系统中采用的则是 IBM 协议。来自苹果公司的系统则在各种各样的硬件拓扑结构上部署运行了称为苹果对话（AppleTalk）的协议。为了在使用上述其他系统的客户的网络中占有一席之地，微软必须得安装得力的网络支撑系统，且该系统可以通过支持上述系统所采用的协议从而实现与相关系统之间的方便的通信。当然，NT 操作系统还必须要提供与早期版本的 Windows 和 DOS 操作系统所采用协议的兼容性。因此，NT 操作系统包含了相关这些系统所使用的所有标准协议。（这些协议有时对多个系统是共用的——例如，VAX 系统也常常采用传输控制协议和网际协议）。相关典型协议包括：

- Novell Netware 所采用的互联网数据包交换协议和序列数据包交换协议
- UNIX 所采用的传输控制协议和网际协议
- 数字设备（Digital Equipment）公司 VAX 系统所采用的数字设备公司分层网络体系结构协议（Digital Equipment Corporation network, DECNet）
- IBM 系统所采用的系统网络体系结构（System Network Architecture, SNA）协议和网

络基本输入 / 输出系统增强型用户接口 (NetBIOS Enhanced User Interface, NetBEUI) 协议

- 传统 Windows 操作系统所采用的局域网管理器 (LAN Manager) 协议

与 NT 操作系统的大多数其他主要组件相类似, 其网络功能也进行了分层处理。例如, 网络栈 (networking stack) 的最底层采用了由微软公司和 3Com 公司所定义的称为网络驱动程序接口规范 (Network Driver Interface Specification, NDIS) 的接口。该接口是为允许单个硬件设备驱动程序支持多个网络层协议而专门设计的。于是, 网络接口卡 (network interface card, NIC; 或简称网卡) 的供应商就可以为网卡和硬件平台的每个组合编写一个单独的驱动程序, 而不用考虑具体的操作系统或网络层软件。实际上, 这便允许有关驱动程序同时支持多个网络层协议。与输入 / 输出系统一样, 这种层次式体系结构允许透明地插入大多数用户并不需要的额外的功能。一个例子是, 通过在个人计算机系统中插入提供简单网络管理协议 (Simple Network Management Protocol, SNMP) 功能的层级, 就能够利用基于简单网络管理协议的网络管理控制台对相应的个人计算机实施远程监控。该协议曾经在第 15 章中进行过讨论。而在此类监控没有需要的情况下, 也不用非得要把相应功能安装到系统上和造成资源的无端浪费。

440

NT 操作系统网络支持的一个很有意思的特性是其包含有一个面向异步传输模式 (Asynchronous Transfer Mode, ATM) 硬件的接口。异步传输模式拥有若干有趣的特征, 但却被大多数人在其匆忙加入以太网浪潮的过程中给忽略了。首先, 网际协议的最大帧大小是 64KB, 同以太网 1500 字节的最大帧大小相比, 异步传输模式 64KB 的最大帧大小与网际协议的最大帧大小更为吻合, 也更适合。特别地, 当硬件可以直接支持较大型的数据块时, 却非得要把它们划分成小型的数据块, 这无疑是对资源的极大浪费。另一方面, 异步传输模式不用求助于特别的软件模块和额外的软件层, 就能够支持硬件中的服务质量 (Quality of Service, QoS) 特性。随着多媒体应用变得越来越重要, 一些人已经发现相关应用系统在异步传输模式上要比在以太网上工作得更好, 为此, NT 操作系统已经包含了对相关功能的支持。

18.9 对称多处理

支持 NT 系列操作系统的硬件平台可扩展到非常大型的系统。而拥有多处理器则是在那些被设计用来支持大容量服务器的系统中经常可以发现的一项特征。目前, 多处理器技术正在向下迁移, 并开始进入普通的桌面系统, 相关桌面系统带有能够并发运行多个进程的处理器的, 而且单块芯片中包含有多个处理器。NT 操作系统支持曾在第 6 章和第 9 章讨论过的对称多处理 (Symmetric Multi Processing, SMP) 系统。NT 系列操作系统支持的最大处理器数量随处理器字的大小而相应变化, 进一步说, 32 位处理器最多支持 32 个处理器, 而 64 位处理器最多支持 64 个处理器。之所以存在这样的限制关系, 只是因为关于各个处理器的掩码是被存储在单个数据字中的。

18.10 XP 操作系统启动速度提升措施

XP 版操作系统的一项令人关注的设计目标是加快引导操作系统的速度及减少启动时间。这一目标在一定程度上取决于系统启动的方式。对于冷启动 (cold start) 来说, 有关目标即启动时间期望值无疑比从待机模式 (standby mode) 或休眠模式 (hibernate mode) 所需的启

441 动时间要长得多。进一步说,从待机状态重新启动,5秒的启动时间就是目标值。请注意,这需要启用一个称为高级配置电源接口(Advanced Configuration Power Interface, ACPI)的硬件选项。这项目标的时间间隔很有意思,因为它大致是人类短期记忆的超时限值。如果你开始执行某项任务,并且启动该项任务所需的操作大约需要7秒钟以上,那么你会经常发现,执行伊始,你的注意力却已经开小差了——例如,你可能忘掉了你刚刚查找过的电话号码。因此,如果你的个人计算机处于关机状态,而你决定打开它来查找某些有趣的东西,且如果它的启动需要7秒多的时间,那么你可能会发现,你曾经闪出的好的想法刚才却悄悄地溜走了。所以这是一项非常重要的功能,尽管许多用户对此可能并不完全理解,但这项功能却的确在影响着他们。

18.11 小结

在本章中,我们讨论了一种比较高级的操作系统——微软公司开发的 Windows NT 操作系统的有关功能和概念。首先,我们对 NT 操作系统进行了概述,并简要介绍了微软操作系统的发展历程。然后,我们扼要地就高端单用户操作系统的性质以及 NT 系列操作系统的主要目标——对源自传统操作系统的应用程序的支持以及对多种硬件平台的支持——进行了论述。接下来,我们讨论了同时运行多用户应用程序和服务器应用程序所带来的复杂性。需要强调的是,这种新增的复杂性在 NT 系列操作系统所支持的无论进程和线程的调度还是额外的存储器管理功能中都有所体现。

在此基础上,我们概要阐明了 NT 操作系统中的文件支持、在有关系统上配置多个用户以及多个服务器所需的更高级别的功能,我们同时还讨论了有关操作系统所提供的输入/输出功能的范畴。其后,我们简要阐述了由于同时打开多个窗口而导致的图形化用户界面功能的一些新的考量,另外我们还提到了 NT 操作系统下的多处理器支持的话题。最后,我们探讨了 XP 操作系统的启动速度。

在下一章中,我们将会围绕 Linux 操作系统展开进一步的实例研究,主要涵盖那些没有在螺旋式教学章节所涉及的功能特征,重点是支持多用户所需的相关功能。

参考文献

IEEE: Information Technology—Portable Operating Systems Interface (POSIX). New York: IEEE, 1990.
Ricadela, A., “Gates Says Security Is Job One For Vista.” *InformationWeek News*, February 14, 2006.

Russinovich, M. E., and D. A. Solomon, *Microsoft Windows Internals*, 4th ed., Redmond WA: Microsoft Press, 2005.

网上资源

<http://www.activewin.com/awin/default.asp> (局外人发布的关于微软的信息)
<http://book.itzero.com/read/microsoft/0507/Microsoft.Press.Microsoft.Windows.Internals.Fourth.Edition.Dec.2004.internal.Fixed.eBook%2DDDU%5Fhtml/> (Microsoft® Windows® Internals, 4th ed. Microsoft Windows Server™ 2003, Windows XP, and Windows 2000, by Russinovich, M. E., and D. A. Solomon)
<http://msdn.microsoft.com/en-us/default.aspx> (微软开发人员新闻)
<http://www.osnews.com> (关于各种操作系统的新闻网站)

<http://technet.microsoft.com/en-us/library/bb878161.aspx> (Windows XP资源工具包)
<http://technet.microsoft.com/en-us/sysinternals/default.aspx> (Sysinternals, 原先是外围的技术参考资料,后来被微软公司所收购)
<http://pages.prodigy.net/michaln/history/> (OS/2操作系统发展史)
<http://www.tasklist.org> (用于列举系统上运行的所有进程的软件)
<http://www.windowsitlibrary.com> (杂志网站)
<http://www.winsupersite.com> (局外人发布的关于微软的信息)

习题

- 18.1 当开发 Windows NT 操作系统的时候, 主要发生了什么样的变化, 使得其不同于微软以往大多数的操作系统产品?
- 18.2 本章提到的关于 XP 系列操作系统的一些主要目标都有什么?
- 18.3 当一个进程执行 fork 调用时, XP 操作系统并不会真正创建相应程序的第二个副本。那么它做了什么呢?
- 18.4 如何解决硬件独立性的目标?
- 18.5 NT 操作系统使用什么类型的对象来调度处理器呢?
- 18.6 请描述普通优先级类别和实时优先级类别之间的区别。
- 18.7 关于 NTFS 对文件中的数据的支持方案, 有什么不同寻常的地方? 特别地, 如果有关数据非常短小, 会发生什么呢?
- 18.8 Windows XP 操作系统是否支持 OS/2 HPFS 文件系统?
- 18.9 NT 操作系统采用软件方式支持了哪些廉价磁盘冗余阵列构型?
- 18.10 为什么在输入 / 输出系统和文件系统之间进行这样一个明确的划分非常重要?
- 18.11 NT 操作系统支持压缩文件甚至文件系统的整个部分。这是否正确?
- 18.12 “基于日志”的文件系统的影响是什么?
- 18.13 可动态安装的设备驱动程序的优点是什么?
- 18.14 Windows XP 操作系统的命令行界面有什么不寻常的地方?
- 18.15 有一派观点认为, 应用程序在其图形化用户界面中使用标准要素更为合适。而另一派观点则认为, 改进的要素能够让应用程序变得更好。请论证你自己的选择。
- 18.16 为什么 XP 操作系统支持异步传输模式协议栈?
- 18.17 网络驱动程序接口规范是做什么的?
- 18.18 XP 操作系统支持哪种多处理机制?

Linux 操作系统实例研究

在本书的第二部分，我们曾经讨论过 Linux 操作系统的一些基本特征以及多用户设计对操作系统提出的一些特别需求。在相应章节，我们还概要介绍了 Linux 及其历史背景，并讨论了多用户操作系统的一般特性、Linux 操作系统中的进程和进程调度以及用户登录的本质和文件保护机制。

在本章中，我们将从操作系统实例研究的角度进一步阐述 Linux 的更多信息，并诠释其是如何实现那些我们期望在任何现代操作系统中所看到的标准功能的。本章着眼于与第 6 章相关素材的互相补充和完善，故而尽量避免不必要的内容重复。在本章的开始部分，我们将首先简要回顾一下 Linux 及其发展历程。接下来，我们在第 19.2 节将讨论 Linux 操作系统的进程调度。在此基础上，我们将在第 19.3 节进一步讨论支持在许多不同进程上工作的诸多用户所需的内存管理特性。第 19.4 节则主要介绍 Linux 操作系统中的文件组织，特别地，鉴于 Linux 操作系统独特的发展历程，所以其支持许多不同的文件系统。在第 19.5 节，我们将阐明 Linux 操作系统提供的基本输入/输出功能，而在第 19.6 节，将对源自于 UNIX 所使用的相关设计的图形化用户接口编程支持展开论述。其后，我们在第 19.7 节将讨论 Linux 系统中的网络支持，就像文件系统一样，由于 Linux 的演化过程及其必须共存的现实环境，所以相关支持也较为复杂。在第 19.8 节，我们探讨 Linux 系统的一些特殊安全考量，在第 19.9 节则讨论 Linux 系统在支持多处理器方面所遇到的问题，而在第 19.10 节则主要涵盖 Linux 操作系统的硬实时操作系统变种和嵌入式操作系统变种。最后，我们在第 19.11 节对全章内容进行归纳总结。

19.1 引言

19.1.1 Linux 发展简史

Linux 操作系统在很大程度上是以 UNIX——一种诞生更早、支持多个用户通过连接到大型计算机的终端来使用计算机的操作系统——为导向的。现在，有些版本的 Linux 被用作单用户的个人计算机上的操作系统，尽管这些版本仍然保持着多用户机制的内部结构。实际上，单个用户可能运行多个虚拟终端，并且可能在它们之间进行切换，就像系统上有多个用户一样，同时有关系统还可以支持来自远程连接的相关用户的并发会话。而某些其他版本的 Linux 操作系统的目的则在于纯粹作为远程的服务器，承担各种各样的功能，譬如，用作网络中的路由器、用来控制实时系统以及嵌入没有人机接口的设备中等。正如前面在第 6 章中所指出的那样，Linux 是以生产版本和开发版本的方式发布的，而本章中所描述的功能特征主要与 2.6 版内核有关。

Linux 操作系统的历史比许多其他的操作系统要短，以下是其比较重要的版本和功能的简短总结：

- V. 1.0, 1994 年 3 月，仅支持单处理器 i386 机器。
- V. 1.2, 1995 年 3 月，增加了对 Alpha (阿尔法)、SPARC (斯巴克) 和 MIPS 处理器

的支持。

- V. 2.0, 1996 年 6 月, 增加了对更多型号的处理器及对称多处理系统的支持。
- V. 2.2, 1999 年 1 月。
- V. 2.4.0, 2001 年 1 月。
 - 惠普 (Hewlett-Packard, HP) 公司的 PA-RISC 处理器。
 - 安讯士 (Axis Communications) 公司的 ETRAX CRIS 处理器。
 - 企业标准架构即插即用 (ISA Plug-and-Play)、通用串行总线 (Universal Serial Bus, USB) 架构、PC 卡和蓝牙 (Bluetooth)。
 - 廉价磁盘冗余阵列设备。
- V. 2.6, 2003 年 12 月 17 日。
 - uClinux (一种嵌入式 Linux 操作系统, 用于没有分页式内存管理部件的机器)。
 - 日立 (Hitachi) 公司的 H8/300 系列处理器、NEC 公司的 v850 处理器、摩托罗拉公司的嵌入式 m68k 处理器。
 - 英特尔公司的超线程和物理地址扩展 (Physical Address Extension, PAE)。
 - (每个系统的) 最多用户数和组数可达 4 294 967 296。
 - 进程标识符的最大数目可达 1 073 741 824。
 - 文件系统最高可达 16TB。
 - 无限带宽 (infiniband) 支持。

19.1.2 内核体系结构

Linux 内核结构非常庞大, 但同时也是高度模块化的, 允许个别子系统非常方便地被实验版本所替换。不过, 各个模块之间的关系错综复杂。事实上, 几乎没有什么模块不以某种方式与大多数的其他主要模块发生交互。图 19-1 显示了一些主要组件以及相关模块之间的最重要的关系。在本章的后续部分, 我们将就一些主要系统模块的有关机制展开讨论。

446

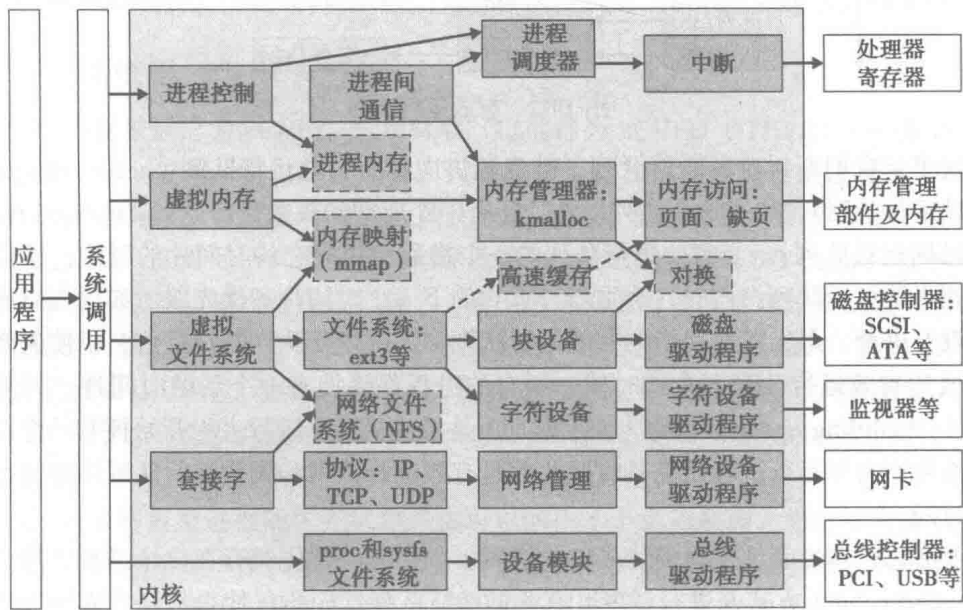


图 19-1 Linux 系统体系结构

19.2 进程调度

在 Linux 内核 2.6 中对进程调度模块 (process scheduler module) 进行了重新设计, 其目的在于构建一个采用 $O(1)$ 级时间复杂度算法的调度器。在先前的内核版本中所用调度器的时间复杂度是 $O(n)$, 当系统负载很高的情况下运行状况表现欠佳。进程调度模块 (SCHED) 负责选定应当由哪个进程来访问处理器。Linux 文档通常使用术语“任务”而不是“进程”, 但对于大多数情况而言, 我们可以认为二者是等同的。Linux 使用基于优先级的调度算法, 从系统中的可运行进程中进行选择。(可运行进程就是等待使用处理器和要运行的进程。)

具体而言, Linux 系统设立有一个运行队列 (runqueue) 由 140 个列表组成, 且每个优先级一个列表, 相关示例如图 19-2 所示。(在多处理器系统中, 对于每个处理器而言, 往往都各自存在一个类似的结构, 不过现在我们将忽略这一点。) 同时设定, 每个列表按先进先出次序进行扫描, 且即将被调度执行的进程添加到运行队列中其相应优先级列表的末尾。故而, 有关调度算法为找到一个进程 (以调度运行) 所花费的时间并不取决于有效进程 (active process, 或称为活动进程) 的数量, 而是取决于优先级列表 (priority list) 的数量。

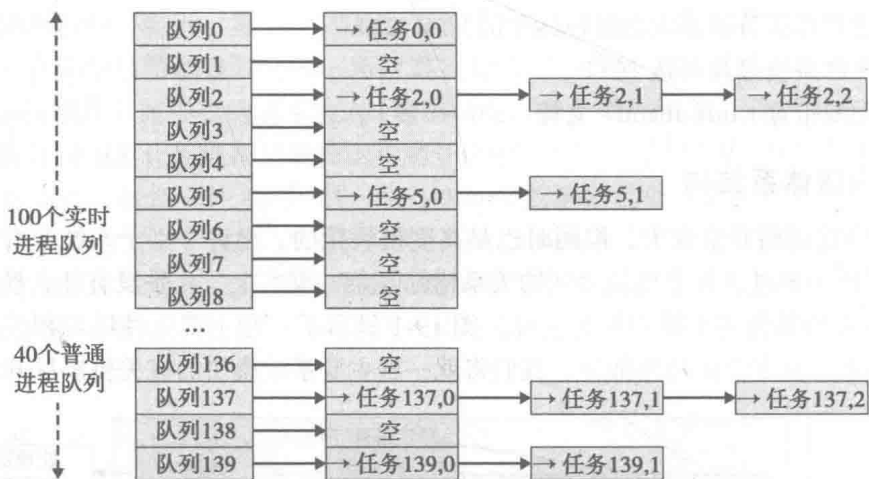


图 19-2 有效运行队列

在这里, 我们所讨论的运行队列, 准确地讲应称为有效运行队列 (active runqueue, 或称为活动运行队列)。除了这个队列以外, 还有一个所谓的失效运行队列 (expired runqueue, 或称为过期运行队列)。当有效运行队列的一个进程使用完它的时间片的时候, 该进程将会被移动到失效运行队列, 同时, 会重新计算它的下一个时间片和优先级。如果在有效运行队列上已没有进程, 那么应对换指向有效运行队列和失效运行队列的指针值, 以使失效运行队列转变成为有效运行队列。在这时候, 所有的进程都将拥有一个新的时间片。另外, 规定调度期 (scheduling epoch, 或称为调度时期) 是指从所有可运行进程开始拥有一个新的时间片, 一直到所有可运行进程用完其时间片并且有效运行队列和失效运行队列需要进行对换之间的时间。

调度器总是调度系统上最高优先级的进程。如果同一优先级存在多个进程, 那么将按照轮转 (round-robin) 方式来进行调度。有关的运行队列结构不仅使得最高优先级进程的查找成为时间恒定 (constant-time) 的操作, 而且还使得优先级内的轮转查找也可能是时间恒定

的。还有，拥有两个运行队列使得时间片调度期之间的变迁也成为时间恒定的操作。

19.2.1 实时进程

标准的 Linux 调度器提供了软实时 (soft real-time) 调度支持，这意味着，尽管该调度器可以很好地满足调度截止时间 (scheduling deadline) 要求，但却不能保证有关截止时间一定得到满足。进一步说，该调度器采用了两种不同的调度类别，以确保所有进程可以对处理器进行公平的访问，同时确保由内核所执行的必要的硬件操作能够按时完成。为此，Linux 操作系统把进程划分成了两类，即普通进程 (normal process) 和实时进程 (real-time process)。具体而言，有关运行队列的前 100 个优先级列表是为实时进程保留的，而最后 40 个优先级列表则可为用户进程所使用。由于实时进程的优先级数值低于非实时进程，所以它们总是会在非实时进程之前运行。（这可能有点令人困惑，因为“较低”的优先级数值却拥有一个意味着将会首先运行的“较高”的优先级，不过，Linux 文档确实就是这样描述的。）只要实时进程是可运行的，那么普通进程就不会运行。实时进程采用两种调度方案来进行调度，即先进先出 (FIFO，对应于宏常量 SCHED_FIFO) 和轮转 (round robin，对应于宏常量 SCHED_RR)。需要作为实时进程运行的进程往往会使用系统调用来告知操作系统应选用哪种调度器。而如果有关进程没有执行这样的系统调用，那么对应进程将会是一个普通进程，具体在下一小节展开讨论。

先进先出进程按照先进先出的方式进行调度。如果有一个先进先出进程已准备好要在系统上运行，那么它将会抢占处理器和取代任何优先级数值更大的其他进程，并且，只要该先进先出进程需要在系统上运行，那么它便会继续运行，因为对先进先出进程没有时间限制。换句话说，多个先进先出进程按照优先级来进行调度，并且优先级数值较小的先进先出进程将会抢占优先级数值较大的先进先出进程所占有的处理器。对于轮转式进程而言，除了具有时间片限制以及它们所占有的处理器往往会被先进先出过程抢占之外，它们与先进先出进程是完全一样的。在给定的优先级上，轮转式进程以轮转方式进行调度。当每个轮转式进程运行完为其分配的时间片时，将会返回到其所在运行队列中相应优先级列表的末尾。

19.2.2 普通进程

非实时进程被标记为 SCHED_NORMAL (以前称为 SCHED_OTHER) ——默认的调度类别。为了防止某个进程一直占用处理器，而使其他也需要访问处理器的进程陷入饥饿状态，Linux 调度器可以动态地改变有关进程的优先级，具体通过提高计算密集型 (CPU-bound) 进程的优先级数 (从而降低其优先级) 和降低输入/输出密集型 (I/O-bound) 进程的优先级数的方法来加以实现。通常情况下，输入/输出密集型进程使用处理器来设置和启动一项输入/输出操作，然后就去等待相应输入/输出完成。而当进程在等待输入/输出操作的同时，其他的进程便可以访问处理器。另外，与用户通信的进程往往会执行大量的输入/输出操作，故而被赋予比非交互式进程 (noninteractive process) 更高的优先级，从而获得更好的交互式响应效果。

输入/输出密集型进程的优先级数最多可以减少 5 个优先级别，而计算密集型进程的优先级数则至多可以增加 5 个优先级别。进一步说，进程将会根据关于其交互性的启发式计算结果来确定是输入/输出密集型进程还是计算密集型进程。而进程的交互性 (interactiveness) 则要根据进程运行时间与其休眠时间的比较来进行推算。一般而言，计算比典型的输入/输

出操作要快得多。鉴于输入/输出密集型进程请求输入/输出操作之后，接下来就会去等待输入/输出操作完成，所以输入/输出密集型进程往往会比计算密集型进程花费更多的时间来等待，也就是说，休眠时间较运行时间长的进程具有较强的交互性，并可认定为输入/输出密集型进程；反之，休眠时间较运行时间不占优势的进程的交互性不强，并可认定为计算密集型进程。

19.2.3 nice 命令及相关系统调用

449

有时会希望以不同于普通默认级别的优先级来运行一道程序。例如，程序有可能提供了一项比交互式用户功能的优先级要低的后台功能（background function）。相反，进程可能正在运行，且需要比正常情况更高的优先级。总的来说，有两种方法可以改变程序的优先级。首先，用户可以使用 nice 命令来运行一道普通级别之外的优先级的程序，其次，程序可以在运行时通过系统调用来改变自己的优先级。关于 nice 命令，原先的设想是用户可以自由地以较高优先级数（故而具有较低的优先级）来运行一条命令。nice 命令具体用法如下所述：

```
nice [-n increment]... [Command [argument]...]
```

-n increment：increment 表示优先级数增量，取值应当在 1~19。如果没有指定，则自动设定优先级数增量为 10。若 increment 大于 19，则优先级数增量将被自动修正为 19。另外，具有管理员权限的用户可以通过使用负增量，譬如 -10，来使命令以高于正常状况的优先级运行。

command：要调用的命令的名称。

argument：调用命令时用作参数的字符串。

另外，进程也可以通过调用诸如 sched_setparam 之类的操作系统函数来更改其自身的优先级，这里的 sched_setparam 是一个可移植操作系统接口（POSIX）标准的函数。当然，也可以使用其他的操作系统调用。在下面给定的示例中，sched_setparam 用来设置与由 pid 所标识的进程的调度策略相关联的调度参数，而参数 p 的解释取决于所选定的策略。如上所述，在 Linux 系统中支持如下三种调度策略：SCHED_FIFO、SCHED_RR 和 SCHED_NORMAL。

```
#include <sched.h>
int sched_setparam (pid_t pid, const struct sched_param *p);

struct sched_param {
    ...
    int
    ...
};
```

19.2.4 对称多处理负载均衡

Linux 操作系统从 2.0 版内核开始支持对称多处理（Symmetric Multi Processing, SMP）架构。我们曾经提到，当一个系统拥有多个处理器时，将会有多个有效运行队列——每个处理器一个有效运行队列。当在对称多处理系统中创建进程的时候，它们被放置在某个处理器的运行队列上。一些进程运行时间很短，而其他进程则可能会运行很长时间，并且操作系统无法提前知道到底哪些进程运行时间长，哪些进程运行时间短。因此，不可能一开始就会以一种均衡的方式把相关进程分配到各个处理器上。为了保持处理器之间的工作负载的均衡，

可以把任务从负载过重的处理器迁移到负载较轻的处理器上。Linux 调度器将会完成此类负载均衡。进一步说,操作系统每 200 毫秒就会检查确认处理器负载是否失衡。如果发生失衡,操作系统将会设法实现负载均衡。显而易见,把进程移动到另一个处理器上的一个负面效应是,刚开始的时候新的处理器中的高速缓存尚不持有相应进程的任何信息。这无疑会使得有效的存储器访问时间临时性大幅上升,然而对比较繁忙的处理器上的负载的减轻将会使这一问题得到弥补。

450

19.3 内存管理

内存管理器 (Memory Manager, MM) 允许多个进程安全地共享机器的主内存 (main memory, 简称主存或内存) 系统。此外,内存管理器支持虚拟内存,从而允许 Linux 系统支持那些所需内存比系统可用内存还要大的进程。内存中存放的未使用的内容 (包括数据和代码) 将会利用文件系统换出到持久性存储器 (persistent storage) 上,而如果以后要再次使用,则会重新换入内存。

Linux 操作系统从一开始就被设计成是独立于其运行的硬件平台。这便引出了关于操作系统中的各种各样的内部数据结构的大小的几个很有意思的问题。Linux 系统必须应对的第一个问题是,可能支撑其运行的机器的基本字长 (word size) 往往会因处理器不同而不同的事实。其他的细节,比如内存页面大小、内存管理硬件工作机制,等等,也可能会有变化。Linux 操作系统试图通过高度的模块化和高度的可配置性原则来处理这些问题。虽然 Linux 是一个庞大内核的操作系统,而不是一个微内核操作系统,但它仍然是高度模块化的,故而,用另一个不同的模块来替换一个模块,譬如内存管理模块,是相当简单和方便的。例如,当研究表明当前所用的机制并没有使用最有效的方法的时候,这样的模块替换就有可能发生,以试图去使用刚刚设计开发的新的机制。而当有关替换实施被证实太过匆忙并实际上会导致比以前版本更差的性能的时候,也可能发生此类模块替换。例如,在 Linux 内核 2.2 版中的页面置换算法,曾替换了此前所使用的算法,但其本身后来却被证明是存在缺陷的。具体而言,虽然该新版页面置换算法在一般情况下可以正常运行,但在某些情况下,性能可能会变得极其糟糕。因此,在 2.4 版内核中,又重新引入了部分的早期机制。类似地,在 2.6 版内核中也引入了前面所描述的时间复杂度为 $O(1)$ 的调度器。

现代 Linux 内核的内存管理器是一个完全的请求分页式虚拟内存管理器。我们在第 11 章曾经非常详尽地讨论过相关技术,所以在这里不再赘述。需要说明的是,对于 x86 处理器而言,Linux 系统采用两级页表,而对于 64 位处理器而言,其则采用三级页表。从理论上来说,分页机制消除了对连续内存分配的要求,但是像内存直接存取 (DMA) 等某些操作,则会忽略分页硬件电路 (paging circuitry),并在传输数据时直接访问地址总线。考虑到这个问题,Linux 系统于是实现了一种称为伙伴系统 (buddy system) 算法的用于分配连续页框的机制。相关页框信息被保存在系统设立的 10 个物理块链表中,且这 10 个物理块链表分别对应包含 1、2、4、8、16、32、64、128、256 或 512 个连续页框的物理块大小。当接收到关于连续内存物理块的申请时,内存管理器将会在物理块链表中查找相应大小或更大的物理块,如果需要,还将对有关物理块实施分割处理。相反,当物理块被释放的时候,内存管理器将会迭代式尝试把空闲物理块对合并成更大的物理块。简言之,Linux 设立了一组单独的对应于低址内存故而适合于直接内存访问操作的伙伴系统链表。

Linux 操作系统还建立了一种单独的称为内存对象集分配器 (slab allocator, 或称为 Slab

[451]

分配器)的机制来处理关于小的内存区域的申请。内存对象集分配器并非是从单个堆中随机地去分配所有的存储请求,而是把内存看作同类对象,譬如进程描述符(process descriptor)的集合。进一步说,内存对象集分配器将会针对所谓内存对象集(slab)的物理块来实施同类对象的分配,有关物理块只保存单一类型的对象。鉴于许多这样的对象的初始化操作比重重新分配还需要更多的时间开销,因此当一个对象被释放的时候,其往往被缓存处理从而方便以后再次作为同一类型的对象加以使用。内存对象集机制并不仅仅限于系统定义的对象。应用程序也可以创建自己的内存对象集链表,并让内存管理器以相同的方式来对它们进行管理。

Linux 操作系统的模块化设计支持在不同的版本或不同的发行版中使用不同的内存管理器。进一步说,鉴于虚拟内存体系结构并不拥有确定的性能,所以仅用于实时版本的 Linux 操作系统的发行版可能必须得避免基于虚拟内存体系结构来进行设计。类似地,嵌入在没有辅助存储器的掌上电脑或微波中的基于 Linux 的系统可以使用更适合于相应硬件配置条件的内存管理器。在第 20 章中,我们将会就展现此类内存管理器的 Symbian 操作系统及其关于分页硬件的一种非常精巧的使用展开进一步的讨论。

19.4 文件支持

每个操作系统设计人员都必须做出的一项决定是文件系统在辅助存储器(通常是磁盘)上的物理和逻辑布局。一般可以采用若干可选的文件系统布局,并且有关差异可能会对操作系统的性能产生巨大的影响。同样, Linux 操作系统的模块化特性在文件支持方面再次得到了淋漓尽致的展现。

19.4.1 标准文件系统

如前所述, Linux 操作系统的原始版本是在 MINIX 系统基础上开发出来的。因此很自然地,最初的 Linux 版本中所使用的文件系统就是围绕 MINIX 文件系统的物理和逻辑布局而设计的。即使在今天, Linux 操作系统仍然支持 MINIX 文件系统布局。由于 Linux 的各种各样的开发人员为自己的 Linux 操作系统版本设定了不同的使用目标,所以 Linux 对许多其他的文件系统布局也提供支持,具体包括 MS-DOS、OS/2、光盘、数字化视频光盘以及其他的(非 Linux 的) UNIX 版本。尽管如此,对于 Linux 硬盘来说,确实存在一种标准的文件系统,即 ext2fs(第二代扩展文件系统)。许多 Linux 系统文档经常会讨论到“Linux 文件系统”,似乎该文件系统就是目前使用的唯一的一种文件系统。然而,开源项目导致的错综复杂在于,开发人员可以随意创建他们喜欢的操作系统的任何变体。这通常是好的,因为其能够鼓励试验和创新。但是,这也可能会成为一个问题,因为其有可能会使初学者的选择变得复杂化。于是,有人认为,从更大的 Linux 用户社区的角度来看,如果只关注少数几种文件系统,那么稀缺资源可能会得到更有效益的利用。表 19-1 给出了已经创建的一些其他 Linux 文件系统。

表 19-1 其他 Linux 文件系统

EXT	扩展文件系统(代替了 MINIX)
EXT2	第二代扩展文件系统
EXT3	第三代扩展文件系统
XFS	硅谷图形公司(Silicon Graphics) IRIX 操作系统之日志式文件系统

(续)

HFS	Mac 操作系统之多级文件系统
EFS	硅谷图形公司 IRIX 操作系统之扩展文件系统
VxFS	Veritas 文件系统
UFS	早期伯克利软件发行版 (BSD) UNIX 文件系统
BSD FFS	BSD UNIX 文件系统
AIX	IBM RS/6000 UNIX 文件系统
JFS	IBM 日志式文件系统
HPFS	OS/2 高性能文件系统
BeFS	BeOS 文件系统
QNX4 FS	QNX4 操作系统之文件系统
AFFS	Amiga 操作系统之快速文件系统
FAT16	微软 DOS 操作系统之文件系统
FAT32	Windows 操作系统之文件系统
ReiserFS	新型 Linux 文件系统 ReiserFS (采用了完全平衡树结构)
Xia	新型 MINIX 文件系统

19.4.2 虚拟文件系统

Linux 操作系统的这种拥有诸多不同但共存的文件系统的想法并不是什么新鲜的东西。从许多大学取得 UNIX 操作系统源码并对之进行“改良”以适应某种特定的局部需要的角度来说, UNIX 也是以类似于 Linux 的方式开发和发展起来的。一种常见的更改情况是, 为 UNIX 设计一种新的文件系统, 以使用来自某些传统操作系统的现有的文件系统, 例如表 19-1 中的高性能文件系统 (HPFS)。为了处理文件系统的这种多样性, UNIX 引入了**虚拟文件系统** (Virtual File System, VFS) 的概念。虚拟文件系统是操作系统内部介于内核系统调用和文件系统之间的一个附加层, 而且它对于应用程序是不可见的, 如图 19-3 所示。这一层的应用程序接口与标准 UNIX 文件系统的应用程序接口完全相同。当通过虚拟文件系统层打开一个文件时, 虚拟文件系统将会查看所引用的设备上的文件系统类型, 并将对应请求传递给该设备的相应文件系统驱动程序。无论是驱动程序还是应用程序都不知道这一附加层的存在。在其他的操作系统中, 有关概念有时被称为**文件重定向器** (redirector)。

这种简单的机制促进了许多原本可能很困难的事情的发展。例如, 当引入只读光盘 (Compact Disc Read-Only Memory, CD-ROM) 设备的时候, 工业界就能够在单一数据文件格式 (音乐光盘遵循另外不同的旧的格式) 基础上进行标准化。其部分原因在于, UNIX (以及其他的操作系统) 已能够在这些设备上支持一种不同于硬盘或软盘上的文件系统格式。这对于工业界和用户群体来说具有极大的好处。想象一下, 如果每种操作系统都有其自己的光盘格式, 那将是怎样一种情形! 你真的不难想象到——其会与以前的软盘行业别无两样。具体而言, 你必须得购买已经按照你即将要使用的操作系统的要求格式化过的软盘, 磁盘可能是固定扇区 (hard-sectored, 或称为硬扇区), 也可能是软扇区 (soft sectored), 并且具有不同的密度 (指每个磁道可以容纳的扇区数量)。至少对于软盘的例子来说, 当微软 DOS 系统流行开来的时候, 其软盘格式便无处不在, 于是各种操作系统不得不以某种方式来

452
453

处理 DOS 格式的软盘，所以情况并没有像预料中那么糟糕。另外，太阳微系统公司（Sun Microsystems）则利用这种机制引入了一种“文件系统”，其实际上是一种访问不在本地磁盘上的文件的网络协议，也就是说，有关文件其实是跨网络驻留的。相关机制的远程特性对于应用程序来说是完全透明的，但是对这项功能的简单使用有时可能会严重地影响系统性能。这种文件系统被称为网络文件系统（Network File System，NFS）。

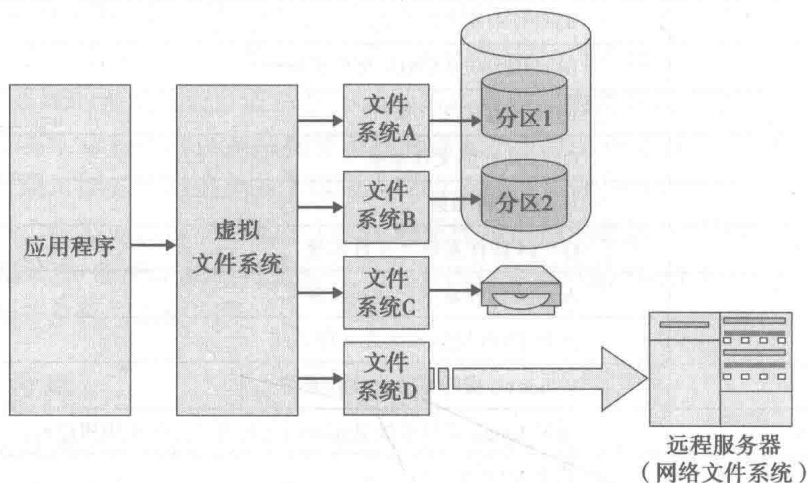


图 19-3 Linux 虚拟文件系统（VFS）

19.4.3 进程文件系统

Linux 操作系统（以及 UNIX 的一些衍生版操作系统）还以一种非常有创意的方式来利用文件系统接口。进程文件系统（proc file system）从其没有引用磁盘上文件的角度来说，并算不上一种真正的物理文件系统。它有时被称为虚文件系统或伪文件系统（pseudo file system），并被认为是非持久性的。它会响应大多数的与任何其他文件系统相同的系统调用，但是并不访问存储设备，而是返回关于操作系统内核相关变量的信息。鉴于根文件系统（root file system）是通过标准输入/输出调用来进行访问的，所以人们经常只是打开一个 proc 表项（位于 proc 子目录中）然后读取对应的内容。有关返回的信息实际上并不以那样的格式存在于内核中（虽然有些部分可能是这样），而是在执行读取操作时动态创建的。相关记录包括关于在系统上运行的进程的信息以及关于诸如网络、内存管理等其他模块的信息。在用户看来，进程文件系统内部甚至还有目录。例如，有一个 /proc/net 目录，其中包含有关网络模块的所有信息。其他目录则往往对应于系统上运行的各个进程。这些目录有时会包含子目录来对应于特定模块的子功能。例如，/proc/net 目录包含有对应地址解析协议表（即 arp 表）的子目录以及对应于传输控制协议和网际协议等网络协议的参数和计数器的子目录。请注意，进程文件系统既支持读操作，也支持写操作，故而内核中的数据可以被小心谨慎地加以更改。通常情况下，这意味着只有具有 root（超级用户）权限的用户才能写入此文件系统。

19.5 基本输入/输出

19.5.1 设备表

设备表（/dev table）即设备“文件系统”，是在 Linux 系统中可以见到的一种独立的类

似于进程文件系统的“文件系统”。Linux 系统上的大多数设备都在 /dev 目录中拥有一个相对应的“文件”，当然，网络设备是个例外。同时，/dev 目录中的每个文件都有一个相关联的主设备号（major device number）和次设备号（minor device number）。而内核将会利用这些编号来实现从设备文件引用到相应的驱动程序的映射。

具体来说，主设备号标识了与对应文件相关联的驱动程序（换句话说，就是设备的类型）。有关编号由 Linux 名称和编号分配机构（Linux Assigned Names And Numbers Authority, LANANA）负责分配和指定。而次设备号则通常用来标识将被寻址的给定设备类型的某个特定的实例。例如，对于硬盘而言，可以有不同类型的硬盘，如小型计算机系统接口（Small Computer System Interface, SCSI）类型和串行高级技术连接（Serial Advanced Technology Attachment, SATA 或串行 ATA）接口类型，而且可能有两个串行高级技术连接接口类型 SATA 的硬盘，后者根据次设备号来加以区分。为此，次设备号有时也被称为部件编号。

通过在 Linux 命令解释器（shell，或称为外壳程序）中输入以下命令，就可以查看设备文件的主设备号和次设备号：

```
ls -l /dev/sda
brw-rw---- 1 root disk 8, 0 Mar 3 2007 /dev/sda
```

该示例给出了 Linux 系统上的第一个小型计算机系统接口类型 SCSI 的磁盘。它的主设备号是 8，而次设备号为 0。次设备号有时被相应驱动程序用来选择设备的某些特殊的特性。例如，一台磁带驱动器可能在 /dev 目录中拥有若干不同的分别用来表示各种记录密度和倒带特性配置的文件。事实上，有关驱动程序可以采用任何其想要的方式来使用次设备号。

请注意，上面的“ls”命令通常用于显示一个目录中的文件，这里却被用来显示设备的特性，就像它是一个实际的物理文件一样。

19.5.2 设备类型

与大多数的操作系统一样，Linux 操作系统大体上也把设备划分为三种类型，即块设备、字符设备和网络设备，并且对每种类型的设备采取了不同的处理方式。图 19-4 显示了 Linux 系统的一些内核输入 / 输出模块以及彼此之间的关系。

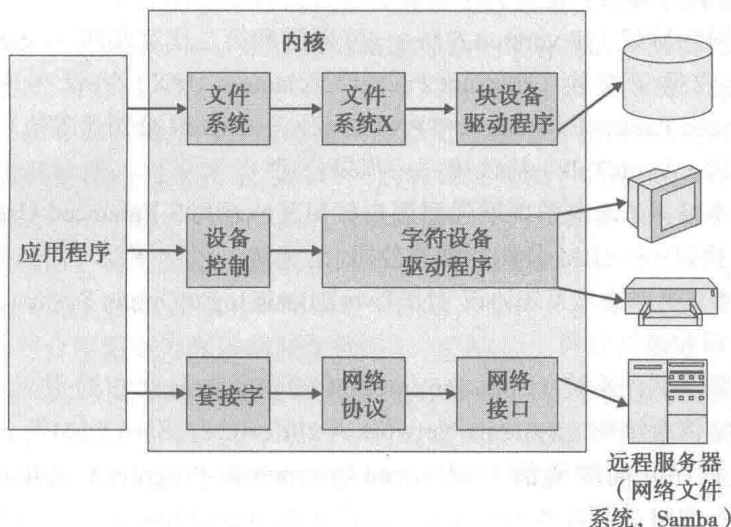


图 19-4 Linux 输入 / 输出系统

块设备

对于 Linux 操作系统来说, 往往通过文件系统来访问块模式的设备 (block mode device, 即块设备), 即使是磁带存储器也是如此。同时, Linux 还支持直接对设备进行原始的输入/输出 (raw I/O) 操作。

字符设备

字符模式设备 (character mode device, 即字符设备) 每次传输一个字节的数据, 包括打印机、键盘、鼠标 (和其他的指针式设备) 等。一般而言, 程序可以使用 ioctl 系统调用访问大多数的字符模式的设备。

网络设备

网络设备不符合文件的语义, 因为等待输入的应用程序永远也不会知道相关输入何时可能到达。因此, 网络设备具有一套与其他设备完全不同的接口。

455 鉴于网络设备的操作是如此得与众不同, 所以它们并没有出现在设备表 (/dev) 中。相反, 它们拥有另外的一套通用的网络接口, 其在大多数情况下遵循传输控制协议和网际协议 (或用户数据报协议) 的协议栈模型, 并且经常利用所谓套接字 (socket) 的编程模型来进行访问。有关编程模型是一种应用程序接口, 允许一个应用程序与另一个应用程序建立网络连接, 有可能但不一定是在不同的系统上, 并且在两个应用程序之间发送和接收一连串数据或一系列数据块。这一模型具有数据链路、网络和传输控制服务等层级。大多数类型的设备驱动程序都保存着关于对应操作 (包括错误) 的各种各样的使用统计信息, 以便系统管理员可以优化系统的性能。网络驱动程序比大多数其他的驱动程序更为积极主动, 通常情况下会保存许多不同类型的统计信息, 包括错误计数器以及发送和接收的数据包数量。相关网络模块通过一套通用的网络接口, 来支持连接、发送、接收、超时处理、统计收集以及路由等公共操作。由于 UNIX 操作系统的起源是与传输控制协议和网际协议的利用紧密联系在一起的, 所以获知 Linux 驱动程序针对传输控制协议和网际协议的支持进行了优化也不足为怪。尽管如此, 再次强调, Linux 操作系统的开放性和 Linux 支持人员的多样化需求导致了許多其他网络协议也针对 Linux 进行了改编处理。如下是可供 Linux 操作系统使用的许多其他协议中的一部分:

- 网络协议 (软件协议)
 - 第 6 版网际协议 (IP version 6) —— 互联网和第二代互联网 (Internet 2)
 - 互联网数据包交换 (Internet Packet Exchange, IPX) 协议和序列数据包交换 (Sequenced Packet Exchange, SPX) 协议 —— Novell 公司 (诺勒)
 - 苹果对话 (AppleTalk) 协议簇 —— 苹果公司
 - 网络基本输入/输出系统增强型用户接口 (NetBIOS Enhanced User Interface, NetBEUI) 协议 —— IBM 公司和微软公司
 - 网络基本输入/输出系统协议 (NETwork Basic Input Output System, NetBIOS) —— IBM 公司和微软公司
 - 通用互联网文件系统 (Common Internet File System, CIFS) 协议 —— 微软公司
 - 系统网络体系结构 (Systems Network Architecture, SNA) 协议 —— IBM 公司
 - 程序与程序间高级通信 (Advanced Program-to-Program Communication, APPC) 协议 —— IBM 公司
 - 数字设备公司分层网络体系结构协议 (Digital Equipment Corporation Network,

DECNet) ——DEC 公司

- 物理协议 (硬件协议或路由器和交换机接口)
 - 综合业务数字网络 (Integrated Services Digital Network, ISDN) 协议
 - 点对点协议 (Point-to-Point Protocol, PPP)
 - 串行线路接口协议 (Serial Line Interface Protocol, SLIP)
 - 并行线路互联网协议 (Parallel Line Internet Protocol, PLIP)
 - 业余无线电联盟数据链路层协议 —— (Amateur Radio - AX25)
 - 异步传输模式 (Asynchronous Transfer Mode, ATM) 协议
 - 附属资源计算机网络 (Attached Resource Computer Network, ARCNet) 协议 —— Datapoint 公司等
 - 光纤分布式数据接口 (Fiber Distributed Data Interface, FDDI) 协议
 - 帧中继 (Frame Relay)
 - 令牌环 (Token Ring) ——IBM 公司
 - X.25 协议 ——慢速、异步方式
 - 802.11 协议 ——无线局域网
 - 蓝牙 (Bluetooth) 协议 ——无线网络

456

为了保持其 UNIX 导向, Linux 操作系统为套接字和数据报机制提供了强有力的支持。有关机制在 1994 年就被包含在 Linux 内核 1.0 版本中, 而这种机制是通过伯克利软件发行版 (BSD) 4.3 版的 UNIX 而被引入 UNIX 操作系统领域的。相关网络接口及对应机制在第 15 章曾进行过较为详尽的讨论, 在此不再赘述。

磁盘调度

在第 14 章, 我们讨论了操作系统在磁盘驱动器上的调度操作的诸多选项, 例如, 何时移动到下一个磁道以及移动到哪条磁道。本章则强调的是作为 Linux 操作系统优势之一的模块化。这种模块化允许用另一不同的实现方案来替换 Linux 操作系统中的个别模块, 因为相应实现方案更适合于特定的情况。关于磁盘操作的调度就是这方面的一个很好的例子。虽然默认的磁盘调度器 (disk scheduler, 或称为磁盘调度程序) 在一般情况下是公平的并且执行得也很好, 但是任何特定系统的总体性能在很大程度上往往取决于正在实施的处理的类型。例如, 万维网服务器 (Web server, 或称为网站服务器) 对系统的要求就与数据库服务器对系统的要求差别很大。因此, 在 Linux 操作系统中同时提供了若干不同的磁盘调度器, 而这些磁盘调度器往往能够比其他未入选的磁盘调度器更好地适合相应的情况。在过去, Linux 系统中默认的磁盘调度器一直采用的是循环向前看调度算法 (Circular-LOOK, C-LOOK)。该算法把磁盘看作是一个圆柱, 从驱动器的一端开始, 在行进中按序处理相关操作。当到达请求操作队列的末端时, 将会把磁头直接移回到另一端且期间不处理任何请求, 然后开始处理在上一轮扫描期间磁头移动过程中所积累起来的请求操作。需要指出的是, 后来的 Linux 发行版本已经开始结合使用更为先进的调度算法。

Linux 系统中磁盘调度器的最新发展状况充分展示了可替换模块是如何被用来获得巨大的好处的。如前所述, Linux 操作系统中的磁盘调度器基本上是一个循环向前看调度 (C-LOOK) 程序, 其仅仅处理在寻道方向上的请求操作。然而有人指出, 这有时会造成那些与其他大部分请求相距很远的磁盘操作请求的调度服务遭受严重不公平待遇的情况。为了改变这种相关请求被“慢待”的状况, Linux 磁盘调度器进行了修改, 即为每个新请求设立了

[457]

一个截止时间。如果某个请求的截止时间即将到达，那么该请求应当立即得到服务。显然，如此改造后的算法在某些情况下能够提供更好的性能。

遗憾的是，执行大量读取操作的应用程序往往倾向于采用同步输入/输出方式。进一步说，通常情况下，有关应用程序一般会先读取一个数据块，接着对该数据块进行处理，然后再发出读取下一数据块的请求。然而，就在对相应数据块处理的同时，磁头已被移动到驱动器的另一位置，而且直到很久以后方可满足刚才的那个应用程序所提出的读取下一数据块的请求。为了改善此类应用程序的磁盘操作性能，Linux 操作系统在 2.6 版中引入了新的**预测式调度器**（anticipatory scheduler）。该调度器基本上采用的还是循环向前看调度（C-LOOK）算法，但是当执行读操作时，将会对把磁头移离当前所读数据块的动作延迟很短的一段时间（也就几毫秒吧），从理论上来说主要是考虑到有关应用程序可能很快就会发出针对下一个数据块的新的读取请求。这种方案在某些情况下例如编译时表现得很好，但在其他情况下，特别是对于交互式任务来说，通常比较糟糕。之所以整体性能较差，应该是由于没有保持寻道机制始终处于忙碌状态而引起的。

为此，另外的一种调度器又被发布出来，称之为**完全公平排队**（Complete Fair Queuing, CFQ）调度器。完全公平排队调度器把同步请求放置到每个进程的单独队列中，同时为每个队列访问磁盘分配相应的时间片。时间片的长短以及队列允许提交的请求操作数取决于分配给相应进程的输入/输出优先级。与此同时，异步类型的请求被批量分到各个优先级的单独的队列中。完全公平排队调度器并不进行预测式输入/输出调度，但通过允许进程队列在同步输入/输出结束时“空转”，故而可能“预见/遇见”相应进程的进一步的输入/输出操作请求，从而为整个系统提供不错的吞吐量。完全公平排队调度器作为 Linux 2.6.18 内核的一部分被发布，并且是该内核版本中默认的调度器。

鉴于不存在对所有情况总能表现最佳的调度器，所以目前 Linux 提供了 4 种可用的调度器：

- Noop 调度器
- 预测式输入/输出调度器（as 调度器）
- 基于截止时间的调度器
- 完全公平排队调度器（cfq 调度器）

人们可以在系统引导时通过设置内核选项 `elevator` 来更改调度器。进一步说，可以把调度器设置为预测式输入/输出调度器（as）、完全公平排队调度器（cfq）、基于截止时间的调度器（deadline）以及 Noop 调度器（noop）中的一种。此外，一些调度器所拥有的某些参数可以在运行时进行调整。

19.6 图形化用户接口编程

当创建 UNIX 操作系统的时候，几乎没有什么计算机的工作是在图形模式下完成的。相反，许多用户往往通过一个基本上也就是个打字机（typewriter）或电传打字机（Teletype）——一个内置有键盘的打印机——的终端连接到计算机上。当阴极射线管（Cathode Ray Tube, CRT）终端开始使用时，它们通常也就显示文本，在很大程度上与当时使用的打印机终端差不多。而支持图形的终端的成本可能与其所连接的计算机几近相同。在这种情况下，UNIX 内核便假定用户界面不是图形化用户界面（Graphical User Interface, GUI，或称为图形化用户接口）。反之，如果想要图形化用户界面，那么必须得采用一种与操作系统内核相分离的

机制来提供。于是，X-Window 系统就被构建用来提供相关机制和实现在 UNIX 操作系统中的图形显示。需要指出的是，X-Window 系统是与平台无关的、用于显示图形的基于客户端/服务器的协议。关于 X-Window 系统相关组件的框图如图 19-5 所示。有关组件的命名在今天看来或许会令人感到困惑——将看图形的系统称为 X-Window 服务器，而把生成图形的系统称为 X-Window 客户端。服务器和客户端有可能运行在相同的机器上，就像它们经常在 Linux 个人计算机环境中的那样，或者它们也可能是通过网络连接而分布在世界两端的不同的系统上。

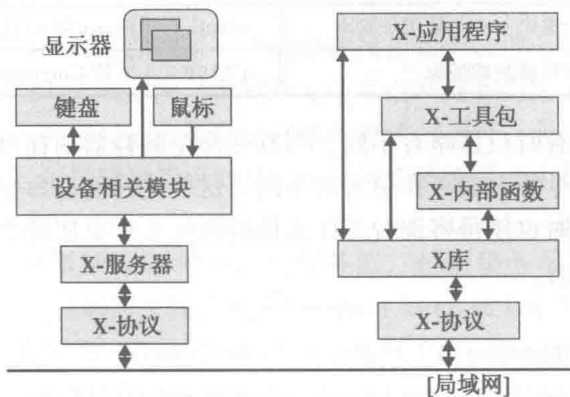


图 19-5 X-Window 系统

X-Window（或 X11）协议需要第三个组件，以按照由管理员所确定的方式来绘制和实际显示窗口、菜单、框和滚动条，有关组件被称为窗口管理器（window manager）。窗口管理器决定了界面的外观以及用户与其交互的方式——即所谓的外观和感觉（look-and-feel）。在单个机器上运行的 X-Window 服务器和窗口管理器通常提供了熟悉的图形化用户界面。目前在 Linux 操作系统领域，主要有两种非常受欢迎的窗口管理器——KDE（Kool Desktop Environment，Kool 桌面环境）和 GNOME（GNU Network Object Model Environment，GNU 网络对象模型环境）。这两种窗口管理器在大多数的 Linux 发行版中都可以找到，相关发行版中偶尔也会包含其他不那么流行的窗口管理器。鉴于 Linux 应用程序通常使用 X-Windows 应用程序接口在系统上进行绘制，所以，与任何一个窗口管理器一起运行的程序往往也可以与任何其他窗口管理器一起正常工作。不过，对话框、菜单、滚动条以及窗口之间的移动可能在用户面前会呈现出不同的显示效果。因此，例如，尽管苹果公司的 X 版 Mac 操作系统是构建在 UNIX 内核上的，但是其窗口管理器看起来和工作起来却更像以前版本的 Mac 操作系统。另外，让窗口管理器作为外部组件的缺点是可能会存在多种不同的图形化用户界面，故而需要为每种图形化用户界面定制相应的教材和培训资料。尽管这对个人来说可能无关紧要，但是对于各种机构来说，则是一个很大的问题，因为它们必须要考虑和支持可能选择不同管理器的许多用户。

对于 UNIX 操作系统而言，这还不是一个仅仅局限于图形化用户界面的问题。传统上，UNIX 命令一般通过在一个文本式命令解释器（command interpreter）或外壳程序（shell，或称为命令解释器）中按行输入来提交给操作系统。UNIX 系统的面向文本的外壳程序是一个单独的外部模块，就像图形化用户界面一样。在 UNIX 系统下，拥有很多类似的命令解释器（如表 19-2 所示）。

表 19-2 面向 UNIX 和 Linux 系统的流行的命令解释器

命令解释器	说 明	命令解释器	说 明
KSH	Linux 版的 Korn 命令解释器	Korn	源自于美国电话电报公司的 System V
TCSH	Turbo C 命令解释器	C	加州大学伯克利分校
BASH	再生版命令解释器	SH	最初的 UNIX 命令解释器
CSH	Linux 版的 C 命令解释器	rc	贝尔实验室 9 号计划
ASH	由 Almgust 原创的 UNIX 命令解释器	es	语法类似于 RC，但采用 Scheme 语义
ZSH	高级命令行编辑——不适用于脚本	eshell	Emacs 完全用 Elisp 实现的类 UNIX 外壳程序
Bourne	最初命令解释器的增强版	CLISP	支持 CommonLisp 的命令解释器

不同的命令解释器有时只是略有不同，但有些命令解释器则在可编程、协助用户补全命令、保存和重用以往命令等方面存在很大的不同。这些差异使得命令解释器的选择成为一个非常个性化的决定，同时也使得咨询台工作人员的技能要求更加复杂化，并限制了一个用户去帮助另一个用户的能力。

19.7 网络

操作系统的网络接口（network interface，NET）模块提供了对一些网络标准和各种网络硬件的访问支持。

19.7.1 网络分层

在 Linux 操作系统中使用的网络模型是建立在标准的传输控制协议 / 网际协议（TCP/IP）模型基础之上的。鉴于物理接口是由网卡（Network Interface Card, NIC，或称为网络接口卡）实现的，所以 Linux 系统一般会忽略物理层，故此有关模型只给出了三个服务层和应用层，如图 19-6 所示。其中的三个服务层也曾经在图 19-4 中进行了描述。通常情况下，操作系统开发人员会采取彼此完全隔离的方式来构建每个网络协议层。而由此造成的结果往往是，相关消息在层与层之间的额外复制所带来的高开销，因为消息从一层到另一层，可能会多出头部或者尾部。Linux 操作系统则通过以所谓的套接字缓冲区（socket buffer，skbuff）为消息分配空间的方法避免此类问题。套接字缓冲区包含有若干指针，分别指向存储整个数据包的连续内存块中的相应位置。当数据从一层传递到下一层时，下层的报头将会被添加到数据中；同样，当数据从下层传递到上层的时候，下层的报头将会被剥离掉。在分配套接字缓冲区的时候，Linux 操作系统将会计算包括数据包所需的各层报头在内的最大长度的内存需求量，并且，刚开始的初始消息被放置在有关缓冲区的中间，以便为来自较低层的报头留出空间。这样，当数据包在层间进行传递时，仅需把相应指针设置为指向相应套接字缓冲区的报头或报尾的新的开始位置即可。



19.7.2 监听连接的超级服务器

Linux 操作系统利用一个称为 inetd 的用于监听连接的超级服务器（connection super-server），来监听被诸如超文本传输协议 HTTP、第三版邮局协议 POP3 和远程终端协议 Telnet 等公共的网际协议服务所使用的众多端口。当一个网际协议数据包到

图 19-6 Linux 网络层次模型

460

达其中的某个端口的时候, inetd 将会启动所选定的服务器程序。对于相关服务不经常使用的情形而言, 这种机制能够更加高效地利用内存资源, 因为特定的服务器程序只有在需要的情况下才会被加载运行。与此同时, 鉴于 inetd 把有关套接字直接连接到了对应服务器进程的标准输入 stdin、标准输出 stdout 和标准错误 stderr 函数上面, 所以在相关应用程序中甚至不需要网络方面的代码。但是, 对于使用较为频繁的协议来说, 往往需要构建和使用一个专用服务器来直接处理有关的服务器请求。

19.7.3 Samba

由于 Linux 操作系统目前是存在于一个由微软操作系统软件所支配的世界中, 所以有大量的精力被投入用来实现 Linux 系统与微软产品的和谐共处和协同工作。前面我们曾简单地提及 Linux 文件系统对当前和以前的微软文件系统格式的支持, 我们还谈到了 Linux 对一些网络协议所提供的支持。不过需要指出的是, 在其间遗漏掉了一个在网络领域占据主导地位的服务器软件包——Samba。

Samba 是实现了许多微软服务和协议的服务器, 包括服务器消息块 (Server Message Block, SMB) 协议、通用互联网文件系统 (Common Internet File System, CIFS) 协议、分布式计算环境 / 远程过程调用 (Distributed Computing Environment/Remote Procedure Call, DCE/RPC)、微软远程过程调用 (Microsoft Remote Procedure Call, MSRPC)、Windows 网络命名服务 (Windows Internet Name Service, WINS) 的服务器 (是一种网络基本输入 / 输出系统命名服务器, 即 NetBIOS Name Server, NBNS)、建立在传输控制协议和网际协议基础上的网络基本输入 / 输出系统协议 (NetBIOS over TCP/IP, NBT)、网上邻居协议 (Network Neighborhood protocol)、包含有 NT 域登录的 NT 域协议 (NT Domain protocol)、安全账户管理器 (Secure Accounts Manager, SAM) 数据库、本地安全授权机构 (Local Security Authority, LSA) 服务、NT 风格打印服务 (NT-style printing service) SPOOLSS、NT 局域网管理器 (NT LAN Manager, NTLM) 以及使用 Kerberos 和轻量级目录访问协议 (Lightweight Directory Access Protocol, LDAP) 的活动目录 (Active Directory) 登录。同时, Samba 还使用这些协议来查看和共享包括打印机等在内的本地资源。

Samba 可以为选定的 Linux 目录 (包括子目录) 建立网络共享。进一步说, 相关目录在微软 Windows 操作系统用户面前呈现为文件夹方式, 而对于 Linux 操作系统用户来说, 则可以挂载对应共享文件系统或者利用文件传输服务 FTP 程序之类的实用程序来访问相应文件。每个目录可以拥有正常的 Linux 文件保护之外的相应的访问权限。另外, 在大多数其他 UNIX 衍生版的操作系统上也可以找到对 Samba 的支持。

461

19.8 安全

19.8.1 Linux 安全模块

关于安全问题, Linux 社区一直存在分歧。有关分歧的症结与安全社区自身关于如何实施安全方面的分歧的事实存在一定关系。关于安全的一项主要考虑是支持高等级的安全往往会牵涉到大量的资源, 显然, 这不仅包括诸如内存和处理器之类的硬件利用因素, 也包括用来构建和正确使用有关安全机制的管理及用户的时间等因素。为此, 如果在特定的系统安装中不需要高等级的安全, 那么就不应当非得耗用相关资源。现在的 Linux 系统的解决方案

是，包含了一种称为 Linux 安全模块 (Linux Security Module, LSM) 的模块。Linux 安全模块由一系列钩子函数组成，而特定的安全实现可以挂到相应的钩子函数上，以便在对象访问时执行授权检查。

当前，有若干不同的安全系统已被设计成是利用 Linux 安全模块钩子机制而运行在 Linux 操作系统上。最知名的是 SELinux (Security-Enhanced Linux, Linux 安全增强模块)，其提供了基于个人请求访问的信任 (许可) 级别的安全访问控制机制。其他的还包括 AppArmor (Application Armor, 应用程序盔甲式访问控制系统)、Linux Intrusion Detection System (Linux 入侵检测系统)、BSD Secure Levels (伯克利软件发行版安全分级系统) 以及 CIPSO (Commercial IP Security Option, 商业网际协议安全选项)。对于 Linux 2.6 版内核而言，这些模块没有哪一个能够对其他模块形成压倒性的优势，因此，相应的 Linux 安全模块方法继续得到了支持。

19.8.2 网络安全

因为 Linux 操作系统的内核源码是免费提供的，所以对于无论安全分析人士还是黑客来说，Linux 通常都被用作首选的操作系统。故此，Linux 操作系统拥有许多工具可用来实施黑客行为和安全保护。

端口扫描程序 (port scanner) 是用来尝试确定目标计算机上正在运行着哪些服务的软件包。一般而言，它们将会尝试建立到达目标机器上的每个可能的端口的连接。根据有关尝试的结果，黑客就可能判断出对应机器是否容易从已知漏洞入手来展开攻击。端口扫描程序可以被设置成是攻击单台机器、一组机器或者某个网络上的所有机器。有许多软件包能够执行端口扫描功能。Nmap (Network mapper, 网络映射器)、SATAN (Security Administrator Tool for Analyzing Networks, 网络分析安全管理员工具)、ISS 和 SAINT 是一些较为知名的端口扫描软件。

462

另外还有**隐式端口扫描器 (stealth port scanner)**。隐式端口扫描器利用低级接口来创建传输控制协议或用户数据报协议的数据包，且有关数据包并不正确遵循相应的协议。例如，一个设置了确认标志位 (ACK) 的传输控制协议数据包往往可能成功通过一个对数据包进行过滤的防火墙，因为有关数据包看起来就像是已建立连接的一部分。如果在与发送方没有建立会话的端口上接收到这样的数据包，那么相关的传输控制协议软件将会以适当的消息予以响应，于是黑客便可以由此推断出目标机器上存在某个进程正在读取相应的端口。

tcpd (TCP Wrapper, 即传输控制协议包装器, 或称为传输控制协议包装程序) 用来针对在 Linux 或其他类 UNIX 操作系统上运行的网际协议服务的网络访问进行过滤。在超级服务器 (如 inetd) 对网际协议服务进行启动的时候，系统会调用传输控制协议包装程序来检查相应连接的来源。进一步说，传输控制协议包装程序支持根据主机或子网的网际协议地址或名称进行过滤。此外，相关网际协议服务也可以被链接到**认证 (ident)** 服务上。后者将会首先查询传输控制协议 (TCP) 端口 113 上的网际协议源地址，并期待得到一条关于标识对应源系统的查询应答。只有在接收到相应应答并且与数据库匹配成功的前提条件下，才会允许建立连接。大多数的网络服务程序可以直接链接和使用有关实现了相应过滤功能的库函数。这种方法可适用于那些没有由超级服务器启动就运行起来的服务，或者是任何处理多个连接的服务。在其他的情况下，只有第一次连接尝试才会通过传输控制协议包装器进行相应数据库的匹配检查。

19.9 对称多处理

在对称多处理系统中，操作系统可以在多个处理器上同时运行。正如第 6 章所提到的，当前正在运行的操作系统的两个（或多个）实例必须通过锁机制的利用来防止它们同时更改相同的数据结构。早期版本的 Linux 操作系统的对称多处理支持采用了单一的锁来实现整个内核的锁定，称之为大内核锁（Big Kernel Lock）。然而，这种机制并不是非常高效，因为在很多时候操作系统的不同实例根本不会操作和处理相同的数据结构。为此，后来的 Linux 内核版本已经开始把对大内核锁的那些引用替换成为对更多的局部锁（localized lock）的引用。当然，对于大内核锁的某些引用仍然在 Linux 2.6 版内核中有所保留，但是多处理器的性能却比以往的内核版本大大地提高了。Linux 内核还把自旋锁用到了一种特殊的读写类型的信号量上，以允许多个读操作进程同时执行读操作，但却只允许一个写操作进程执行写操作，而且相关结构在大多数情况下主要执行的是读操作。例如，网络设备表很少会发生改变，但其读取操作却十分频繁，因此允许多个读操作进程同时读是大有好处的。于是，当需要对网络设备表进行更改的时候，可以在仅由一个写操作进程执行更改操作的同时，短暂地不准许所有的读操作进程执行读操作。显然，这并非经常会出现的情况，也不会对系统性能造成多大的不利影响。

19.10 其他 Linux 操作系统变种

由于 Linux 操作系统内核源码很容易就可以得到，所以存在若干出于某些特殊目的而构建起来的 Linux 衍生版操作系统。其中特别是集中在实时应用领域的实时 Linux 操作系统版本以及面向有限资源的小型系统中的嵌入式 Linux 操作系统版本。

19.10.1 实时 Linux 操作系统

通常的 Linux 操作系统并不是硬实时操作系统（hard real-time OS），其实对于大多数的操作系统都是如此。硬实时系统应当确保满足实际的截止时间，例如，某个进程或线程应当在接下来的 50ms 内运行。硬实时系统排除了传统操作系统演化发展而形成的诸多机制，特别是那些使用随机技术和试图为各个进程公平地提供服务的相关机制。硬实时系统需要为事件及所请求的服务设立截止时间，而正常的操作系统机制不会允许我们提供这样的截止时间支持。嵌入式应用也很重要，其间计算机更像是一台设备中的控制部件，而不是通用的计算工具，此类嵌入式应用通常也是实时系统。这意味着，嵌入式 Linux 操作系统的实现和实时 Linux 操作系统的实现之间存在着相当大的重叠。

关于如何在 Linux 操作系统内核的基础上构建实时系统的问题，主要存在两派思路。首先，某些实时 Linux 操作系统的实现利用一个小型的实时操作系统（Real-Time OS，RTOS）作为主机操作系统，同时让某一版本的 Linux 内核作为单个线程以后台或空闲进程级别的优先级运行在主机操作系统内核上面。换句话说，当没有任何实时进程运行的时候，任何正常的 Linux 应用程序都可以在系统中运行。这种模型被称为 RTLinux。另一派也使用 Linux 内核，但对其进行了大幅度地修改，以使其仅仅包含允许支持实时应用程序接口的相关调度机制。有关机制至少应当包括进程调度器和磁盘调度程序，可能还有网络协议栈。这种实时应用程序接口（Real-Time Application Interface）模型也称为 RTAI。自然地，除了这两种方法之外，还有其他的一些无法融入这两种类型的令人关注的方法。毋庸置疑，要决定某个项目

应当选用哪一个正确的软件包可能相当复杂。幸运的是，已经有一套开源的实时 Linux 通用应用程序接口（real-time Linux common API），其允许程序设计人员在使用无论 RTLinux 还是 RTAI 模型时，均可按照该通用应用程序接口来进行编码。此外，表 19-3 列出了 Linux 操作系统的一些嵌入式和实时系统的实现版本。

表 19-3 常见的 Linux 衍生版操作系统

商业平台
FSMLabs 公司：RTLinuxpro——RTCore，一种让 Linux 作为空闲线程运行的硬实时平台
Lineo Solutions 公司：uLinux——硬实时 Linux 内核，针对消费者电子设备
LynuxWorks 公司：BlueCat（蓝猫）——时序要求严格的中断处理及其他硬件操作控制，以线程方式实现 Linux 内核
MontaVista Software 公司：Linux 的实时解决方案——面向嵌入式和实时应用的 MontaVista Linux，包含有一个可抢占式的 Linux 内核
Concurrent Computer 公司：RedHawk（红鹰）——面向多处理器系统的基于 Linux 的 RTOS 实时操作系统内核，使用了处理器屏蔽（CPU shielding）技术，处理器可以被指定为把 Linux 反锁在外的模式，从而使硬实时进程运行在受到保护的处理器上，并确保中断的响应时间
REDSonic 公司：REDICE-Linux——具有额外的抢占时刻点的实时 Linux 内核，提供实时应用程序接口 RTAI 支持和服务质量（Quality of Service，QoS）保证
TimeSys 公司：TimeSys Reservations——利用动态安装的内核模块对一个 Linux 实时操作系统 RTOS 进行了扩展。同时为特定的进程或进程集保留有固定数量的处理器和网络带宽
开源实现
Accelerated Technology——为嵌入式开发人员提供实时操作系统（RTOS）
ADEOS——提供了一个硬件抽象层，允许实时内核和通用内核共存。支持双内核的硬实时 Linux 环境（譬如无技术专利的 RTLinux 或 RTAI）
ART Linux——Linux 2.2 内核的实时扩展版
Flight Linux——面向航天器使用而设计的实时 Linux 衍生版
KURT（The KU Real-Time Linux）——在堪萨斯大学开发的实时 Linux 内核
Linux/RK——对 Linux 进行了基于可加载内核模块的“资源内核”功能增强，可为应用程序提供及时、有保证和强制的对系统资源的访问。开发基地在卡内基梅隆大学
OnCore’s Linux for Real-Time——允许嵌入式设计人员选择适合相应问题的内存占用和性能模型
RED-Linux——Linux 的实时版本；总部设在加州大学尔湾分校
RTAI——实时应用程序接口，一种可适用于单处理器和对称多处理系统的综合的实时应用程序接口。允许从用户空间控制实时进程 - 利用细粒度进程调度的软实时。AtomicRTAI 是它的一个小型（单软盘）版本
RTLinux——“硬实时”微型操作系统，以其最低优先级且可抢占用户线程方式来运行 Linux，因此实时线程和中断处理程序决不会被非实时操作延迟。支持用户级实时程序设计。MiniRTL 是它的一个小型版本
RedIce Linux——RedIce Linux（红冰 Linux）允许同时运行 RTAI 和 RED-Linux，通过完整的 Linux 内核支持，使极低延迟要求的实时任务和硬实时用户应用程序可以在同一结构下运行

19.10.2 嵌入式 Linux 操作系统

Linux 操作系统内核也被修改用来运行在资源受限的平台上。此类平台包括在 Palm 操作系统相关章节中所讨论的那些平台，同时也包括像微波炉和家庭供暖控制器等用户环境非常受限的设备。Linux 的开源和模块化特性使其成为此类场合的理想选择。然而，在这样的系统中使用 Linux 作为操作系统必须要解决许多问题：

- 首先, 相关设备通常没有辅助存储器, 故而实际上并不需要分页内存。但是标准的 Linux 操作系统往往假定存在辅助存储器。因此, 在嵌入式 Linux 系统中经常见到的一项修改就是去除分页内存硬件支持需求。同时, 相关系统常常也不需要缓存管理系统或者如内存映射文件之类的功能。
- 其次, 极其受限的用户界面也值得注意, 因为 Linux 操作系统并没有把图形化用户界面作为首要需求。相反, 图形化用户界面是一个附加组件, Linux 系统的标准接口是命令行。而对于有关显示装置可能只是仅仅能够显示几位数字的液晶显示器面板的微波炉来说, 甚至这样的假设都可能有点过分。当然, 还有一些嵌入式 Linux 系统存在于诸如小型路由器之类的复杂设备中, 而相关设备现在通常支持超文本传输协议, 并且可以通过远程浏览器来进行管理。
- 通常情况下, Linux 操作系统中的进程调度采用“交互性”(interactiveness)的概念来提升有关与用户交互的进程的优先级。但在嵌入式系统中, 并不存在精心设计的交互式用户界面。常常可能只有一个小的显示屏和几个按钮, 利用一些实时进程或普通的中断处理程序便足以管理得非常到位。因此, 进程调度器可以是采用未引入动态优先级改变的精简版本。

464
465

19.11 小结

作为实际操作系统实例研究部分的一章, 本章继续展示有关系统是如何实现标准的操作系统功能的。进一步说, 本章重点讨论了多用户操作系统 Linux 的相关功能。具体而言, 我们首先讨论了进程调度机制, 接下来从支持潜在的诸多用户创建各种不同的进程的角度出发, 概要说明了有关操作系统虚拟内存管理的基本框架。然后, 我们概述了 Linux 操作系统中的文件支持以及由于 Linux 系统的不寻常的发展历程而支持的各种不同的文件系统, 接着我们还阐述了有关操作系统所提供的输入/输出功能。其后, 我们依次简要论述了 Linux 操作系统中图形化用户界面实现、网络支持和安全保护以及多处理器支持的相关话题。最后, 我们讨论了一些 Linux 衍生版的操作系统, 而相关系统版本的形成往往是立足于那些通常情况下 Linux 等操作系统不常见到的诸如硬实时和嵌入式之类的环境。

参考文献

- | | |
|--|--|
| Bovet, D. P., and M. Cesate, <i>Understanding the Linux Kernel</i> , 2nd ed., Sebastopol, CA: O'Reilly & Associates, Inc., 2003. | Stevens, R., <i>Advanced Programming in the UNIX Environment</i> . Boston, MA: Addison-Wesley, 1992. |
| Gorman, M., <i>Understanding the Linux Virtual Memory Manager</i> . Upper Saddle River, NJ: Prentice Hall, 2004. | Stevens, R., <i>Unix Network Programming</i> . Upper Saddle River, NJ: Prentice Hall, 1990. |
| Love, R., <i>Linux Kernel Development</i> . Indianapolis, IN: Sams Publishing, 2004. | Yaghmour, K., <i>Building Embedded Systems</i> . Sebastopol, CA: O'Reilly & Associates, Inc., 2003. |

网上资源

<http://www.linux.org> (Linux 内核开发之家)
<http://www.kernel.org> (史上重要的内核源码库)

<http://www.tldp.org> (Linux 文档化项目)

习题

- 19.1 对于 Linux 2.6 版内核而言, 重新设计进程调度器的具体目标是什么?
- 19.2 在 Linux 调度器中, 实时进程队列是以先进先出方式进行调度的。这是否正确?

- 19.3 简要描述对称多处理负载均衡。
- 19.4 Linux 操作系统使用了标准的请求分页虚拟内存管理器。这是否正确？
- 19.5 什么是伙伴系统？
- 19.6 Linux 操作系统仅仅提供了一个从 MINIX 派生来的简单的文件系统。这是否正确？
- 19.7 Linux 操作系统的主要贡献之一是其使用了在其他操作系统中所没有的独特的磁盘调度算法。这是否正确？
- 19.8 Linux 操作系统是如何为属于不同用户的文件提供保护的？
- 19.9 在 Windows NT 系列操作系统中，图形化用户界面是操作系统内核所固有的。那么，在 Linux 操作系统中，又是如何提供图形化用户界面的？
- 19.10 在通过网络体系结构的各层之间传递消息时，Linux 操作系统如何防止过多的缓冲区复制？
- 19.11 在多处理器系统上，Linux 内核可以同时多个处理器上执行。当内核必须进入临界区时，它不能调用和召唤“操作系统”停下来等待，因为它本身就是操作系统。那么它是如何做的呢？
- 19.12 实时操作系统具有某些被大多数操作系统所忽略的时间安排要求。目前存在许多商业的和开源的 Linux 衍生版操作系统。我们描述了两种不同的方法来支持 Linux 系统的实时需求。其中，一种方法涉及大幅度修改 Linux 内核的进程调度模块，而另一种方法则在概念上更为清晰和简单。请简要描述相关方法。
- 19.13 什么是大内核锁？它遭遇到了什么现实状况？又是如何应对的？

Palm 操作系统实例研究

20.1 概述

在第 4 章, 我们曾讨论了 Palm 操作系统的功能特征和基本组成。不过, 那些讨论主要限于当时我们研究该操作系统相对于以前所研究操作系统的更复杂的设计目标而面临的问题。本章基本上阐述的是第 5 版之前的 Palm 操作系统。该系统代表了本教材在第二部分所描述的操作系统体系中某个特定的层次位置。故而, 关于这种操作系统, 可以加以论述且尚未在第 4 章做过介绍的内容已不算很多了。为此, 在我们按照与其他两个实例分析相类似的脉络依次展开各节内容的同时, 我们将着眼于提供一些与第 4 章不相关的细节。同时, 考虑到当前该操作系统的不断发展演化, 也会加入一些有助于实现该操作系统在计算机产业界中进行定位的其他材料。此外, 我们还将就此类平台的程序设计以及产业界正在开发的应用系统的发展趋势展开讨论。

在这一章, 我们将首先在第 20.2 节简要重述 Palm 操作系统设计的应用场景及环境类型。然后, 从第 20.3 节到第 20.5 节将简要概括第 4 章的相应的要点。接下来, 在第 20.6 节, 我们将讨论 Palm 操作系统在输入 / 输出子系统领域超越我们曾经在第 4 章所介绍基本功能的若干进展, 它们形成了新型手持式平台的典型的功能特征。其后, 在第 20.7 节和第 20.8 节中, 我们还是就第 4 章的相应内容进行整理和概述。在第 20.9 节则阐明对于受限环境进行程序开发所需要的交叉开发系统的主要类型。另外, 个人数字助理、手机、多媒体播放器原先属于不同类型的设备, 然而, 这些平台目前正在进行整合。因此, 在第 20.10 节将讨论在相关平台上的软件的演化发展, 同时也会涉及 Palm 操作系统在更高版本的一些进展。最后, 我们在第 20.11 节对全章内容进行归纳总结。

469

20.2 多进程操作系统环境

在第 4 章, 我们讨论了 Palm 操作系统及其运行环境。但为了方便起见, 我们在这里简要回顾一下。Palm 操作系统专为一个非常特殊类型的环境而设计。这种环境具有几方面的特点, 在一定程度上将会制约和影响有关操作系统的设计方案。表 20-1 中罗列了相关环境的若干主要特征。

比较小的屏幕尺寸 (screen size) 对 Palm 操作系统的设计影响尤为突出。其意味着用户每次只能与一个程序交互, 而应用程序则被假定为相应窗口始终填满整个屏幕 (主窗口前面可能弹出的小提示框除外)。

尽管后来的模型有时会包含磁盘驱动器, 主要是作为一项附加的功能。但是, 最初的机器并没有包括此类设备, 所以有关操作系统被设计成假定所有程序都驻留在主内存的情况。同时, 缺少键盘意味着操作系统必须得提供手写识别, 对应功能是一个实时应用程序, 故而是一个实时的内核构成了 Palm 操作系统的基础。然而, 用户应用程序并不是实时的, 而且是单线程的。最终形成的操作系统的一系列设计选择结果如表 20-2 所示。

表 20-1 Palm 平台的与众不同的特性

小屏幕尺寸
没有辅助存储器
没有硬盘盘——触摸屏
功率有限，主要为了延长电池寿命
处理器速度较慢，目的在于降低功耗
有限的主存空间

表 20-2 Palm 操作系统的与众不同的特性

程序永远不会停止
没有请求分页（虚拟内存）或磁盘缓存
单窗口图形化用户界面
多种文本输入选项
实时操作系统任务、非实时应用程序
没有多线程应用程序

470

20.3 Palm 进程调度

20.3.1 实时任务

Palm 操作系统的进程调度程序采用抢占式多任务优先级调度策略。它将动态地确定哪一个就绪任务具有最高优先级，同时将会中断一个不太重要的任务的运行，而去执行一个已经准备就绪的更重要的任务。底层的操作系统是一个支持手写识别（handwriting recognition）的实时内核。但是，用户应用程序是无法访问这些功能的。手写识别划分为两个部分：手写笔跟踪（stylus tracking）和字符识别（character recognition）。手写笔跟踪利用标准中断机制来处理源自于手写笔的中断。当手写笔跟踪确认手写笔改变了方向、停了下来或者不再触及屏幕（即离开屏幕）时，就会计算一个用来描述相应移动操作的向量，并将此消息传递给另一个正在运行的任务——字符（涂鸦）识别器。如果该例程可以识别有关字符，将会给操作系统传递消息，以便让操作系统决定如何处理相应字符。通常情况下，该字符最后将被传递给持有焦点且焦点放置在用户输入文本所在的当前窗体的控件上的那个应用程序。当然，有关字符也可能是一个控制字符，而不是文本字符，于是应用程序将会得到关于对应事件的一条消息。如果手写笔触及屏幕的位置不在涂鸦区域内，那么操作系统必须要检测触及点是否在窗体按钮上或者将有关信息传递给对应的应用程序——比如说，或许这就是一个绘图工具软件。

20.3.2 其他任务

只有一个单用户的应用程序持有焦点，而且有关应用程序很可能正在等待用户输入，正如刚才所描述的那样。但是，在正常情况下，Palm 系统往往会拥有后台通信功能来执行诸如电话、数据库同步、蓝牙到耳机等本地设备的连接以及网络浏览和电子邮件之类的互联网访问。此外，特定的用户功能，比如搜索名称，将会触发在当前没有持有焦点的应用程序范围内的一次搜索功能。一般来说，没有持有焦点的任务将会运行在一个事件循环中，以等待有关要求其执行某种操作的事件的信号。还有，Palm 调度程序应确保每个应用程序都能获得时间来完成相应的工作。

20.4 Palm 内存管理

对于 Palm 操作系统而言，相关进程始终驻留在主内存（primary memory，简称主存或内存）中。一个进程一旦开始运行，就永远不会真正停止。期间，它可能会失去焦点，从某种意义上来说，它不再运行。但是，它仍然存在于相应的等待再次被从菜单上选中进而恢复执行的队列中。

Palm 操作系统的内存管理器把一大块主存作为堆来进行处理。当内存在堆上进行分配和回收时,最终往往会造成许多的外部碎片。这便需要偶尔的紧凑处理,从而将碎片聚集成比较大的内存块。为了方便有关操作,内存中相关各项内容通过一张内存指针表 (Memory Pointer Table, MPT) 来间接寻址。故而,当内存管理器移动某项内容时,仅仅需要更新对应的内存指针表表项并使其指向该项内容即可。有关机制具体曾在第 4 章进行过详细介绍,在此不再赘述。 [471]

20.5 文件支持

在 Palm 操作系统程序设计人员的文档中常常会提到“数据库”,但实际上就是随机访问的平坦型文件。它们通过一个 16 位整数的“索引”值来进行访问。相关记录是可变长度的,并且可以动态地调整大小、添加和删除。有关文件管理器维护着每个数据库的索引,给出相应数据库中每条记录在内存的当前位置。需要强调的是,一个数据库必须完完整整地存储在单块存储卡内。

20.6 输入 / 输出子系统

早期的 Palm 设备仅仅被用作个人数字助理。正如曾经提到的那样,经过不断的演化和发展,这些系统已经拥有了游戏、手机、万维网浏览器 (Web browser, 或称为网络浏览器)、媒体播放器 (media player) 等许多附加的功能。客观地说,最初版本的 Palm 操作系统功能确实十分有限,特别是在音频方面,然而,相关方面通过各种版本的不断改进已经得到了显著的提高。此外,从通信和网络功能的进步也可以看出有关平台的发展变化。本节将讨论音频功能,而网络功能我们将会放在后面的小节来进行介绍,以保持与其他实例研究章节结构的一致性。

20.6.1 音频输入 / 输出

技术的持续发展使人们对便携式音乐设备的兴趣越来越浓厚。同时,用户希望在这些机器上拥有的高级游戏往往也需要增强的音频功能,特别是音频输入 / 输出 (Audio I/O) 功能。最初, Palm 操作系统对音频的支持仅仅限于长度较短的警报声和游戏中的一些噪声。在后来的版本中,则添加了乐器设备接口 (Musical Instrument Device Interface, MIDI) 的低级实现,这样就可以由应用程序去创建更加精致的乐声。于是在此基础上产生了许多有趣的应用程序,譬如乐音发生器 (tone generator) 和节拍器 (metronome), 以支持音乐演奏人员把 Palm 操作系统设备当作一个简单的音乐工具。该平台的后期版本还增加了更先进的声音支持,允许相应设备作为手机以及进行音乐文件的播放。再后来,还增加了录音和回放的功能——录制和回放自己的声音——以及音频录制和将音频发送到其他的手机上。

20.6.2 流输入 / 输出

为了方便程序的设计, Palm 操作系统还包含有对文件流 (file stream) 的支持,类似于大多数 C 语言库所提供的标准输入流 (stdin) / 标准输出流 (stdout) 功能。相关功能使用了前面在 20.5 节所讨论到的数据库结构,但允许一些应用程序可以更加方便地移植到 Palm 操作系统上。 [472]

20.6.3 内存型磁盘驱动程序

有关底层操作系统是按照通常没有辅助存储器的嵌入式系统的应用场景而设计的。但是,许多应用程序则是按照从标准文件系统风格的接口运行而开发的。为此,AMX 操作系统(译者注:一种实时操作系统,在这里用作底层操作系统)配备了一个预定义的内存型磁盘(RAM disk,或称为随机存取存储器型磁盘或 RAM 磁盘)驱动程序,利用一部分随机存取存储器来模拟磁盘驱动器。这样就能够使 Palm 操作系统轻而易举地来定义一个作为 DOS 格式化软盘驱动器的随机存取存储器驱动器(RAM drive),从而使应用程序可以更方便地移植到 Palm 操作系统上,而且程序员也无须为了有关系统的使用而专门去学习一个单独的接口。

20.6.4 照相机

伴随低成本、高分辨率的互补金属氧化物半导体(Complementary Metal Oxide Semiconductor, CMOS)图像传感器技术的发展,如今许多的手机和个人数字助理都包含了相机(camera,或称为照相机)部件及功能。同时,鉴于现在有了更大容量的内存,所以有关相机不仅可以拍摄静态图像,甚至还可以录制视频文件。因此,相关设备目前除了能够传输音频文件,还能传输图像文件和视频文件。

20.6.5 通信电路

当前,随着蓝牙和 802.11 无线保真(Wireless Fidelity, Wi-Fi)通信电路的上市推广,最新版的 Palm 设备已经包含了相应模块和功能。它们不但允许其他同步的途径,而且还推动了个人数字助理和手机设备类型以及互联网接入设备的相互融合,比如黑莓(Blackberry)手机就是例证。

20.7 图形化用户接口编程

关于 Palm 平台的图形化用户接口环境,我们在第 4 章已经进行过广泛的论述。辨别有关平台的主要因素就是屏幕尺寸小且内存有限。因为屏幕比较小,所以 Palm 系统并不支持应用程序窗体(form)的重叠。(Palm 操作系统使用术语“窗体”来称谓普通的窗口。)然而,这个平台允许从单个应用程序中触发弹出框(pop-up box)的显示。不过,接下来在对应的应用程序继续运行之前,必须先要关闭这些弹出框(Palm 操作系统称这些框为“窗口”)。同时,Palm 系统有限的内存空间决定了可以由 Palm 应用程序创建的特定的窗口的开发模式,也只能是仅仅填写特定的数据结构和调用操作系统例程而已。其后,当用户选定某个选项时,有关操作系统将承担显示窗口和关闭窗口任务。

20.8 网络编程

20.8.1 个人数据同步

当然,便携式设备应当对通信协议给予强有力的支持。鉴于 Palm 设备一开始的设想是作为个人数字助理,所以其最重要的通信应用就是与个人计算机之间的同步,这样手持设备的数据就能够实现备份。故而,Palm 操作系统提供的初始接口是关于串口(serial port)、红外端口(infrared port)以及通用串行总线端口(USB port)的低级别的驱动程序。同时,还

提供一个更高级别的接口，用于编写应用程序来同步个人信息，譬如联系人列表（contact list）和约会日历（appointment calendar）。为此，组建了一个利益相关的供应商联合会，称之为 Versit Consortium（Versit 联合会）。他们定义了一套关于个人数据交换（Personal Data Interchange, PDI）的标准，包括电子名片（electronic business card）交换格式标准 vCard 以及电子日历和日程安排交换格式标准 vCalendar。现在，这些标准由互联网邮件协会（Internet Mail Consortium）来负责维护。Palm 操作系统还包含有一个库，用来支持应用程序以读者或写者方式打开个人数据交换流，以方便有关同步功能的应用程序的开发。

20.8.2 其他数据同步

一些用户经常关注于定制式应用程序的开发，且有关应用程序往往超出了传统的个人数字助理设备上的应用程序的范畴。进一步说，他们可能拥有特定的数据文件，并需要在 Palm 应用程序和其他平台上的类似应用程序之间进行同步。为此，Palm 操作系统提供了交换库（exchange library），作为所谓交换管理器（Exchange Manager）的操作系统模块的插件。这种机制支持 Palm 操作系统中的应用程序在导入和导出数据记录时无须关心相应的传输机制。例如，一个常常提供给 Palm Powered 手持式设备的交换库就实现了红外开发商协会（Infrared Developers Association, IrDA）的协议，即红外对象交换（Infrared OBject EXchange, IrOBEX）协议。这样便允许应用程序通过红外线方式把对象从一个 Palm Powered 手持式设备播送到另一个 Palm Powered 手持式设备上。对于其他的硬件端口和其他的协议，也存在类似的交换库，譬如短信服务（Short Message Service, SMS）库、电子邮件协议以及蓝牙库等。

20.8.3 互联网应用程序

在过去的几年中，互联网已经非常普及，甚至于几乎是强制性的要求手持式设备能够访问其间可以发现的许多流行功能，特别包括万维网（World Wide Web, WWW）网站以及我们刚才提到的电子邮件协议。于是，更多的协议栈和应用程序接口被添加到了 Palm 操作系统中来支持网络应用程序及互联网应用程序。第一项被加入的是被广泛使用的众所周知的低级的伯克利套接字（Berkeley socket）应用程序接口。该接口允许程序设计人员利用各种协议连接到其他系统上的相关服务上，而无须在应用程序中实现相应协议。该接口的包含使得 Palm 操作系统可以支持传输控制协议（面向连接的）通信或者用户数据报协议（无连接的）通信。

Palm 操作系统支持的第二级协议包括对应用层协议的支持，譬如用于万维网的超文本传输协议（HyperText Transport Protocol, HTTP），Palm 设备的万维网浏览器和万维网服务应用程序使用到了这一协议。当开发 Palm 设备的浏览器时，有一些值得关注的问题需要解决，因为最初几乎没有网站是按照手持设备的非常小的屏幕空间的设想而开发的。尽管如此，当前的超文本传输协议属性支持万维网服务器来判定浏览器是否运行在移动平台上，并进而调整相应的输出样式来适应屏幕的大小。

474

20.8.4 电话应用程序

在第 20.10 节，我们将讨论目前的个人数字助理设备与移动电话之间的融合。Palm 系统的电话管理器（Telephony Manager）提供了一组允许应用程序访问各种电话服务的功能函

数。电话应用程序接口（telephony API）按照所谓服务集（service set）的分组对这些函数进行组织。每一个服务集包含一组相关的功能函数，这些功能函数有可能在某个特定的移动设备或网络上能够支持，但到了另一个设备或网络上就不再支持。其中还有一个应用接口函数是用来支持有关应用程序去确认当前环境中是否支持给定的服务集的。一些比较常见的服务集如表 20-3 所示。

表 20-3 电话应用程序接口服务集

服 务 集	功 能
基本功能服务集	始终提供和可用的功能
配置功能服务集	包括短信服务在内的手机配置
数据功能服务集	数据呼叫处理
紧急呼叫功能服务集	紧急呼叫处理
信息功能服务集	有关当前手机的信息检索
网络功能服务集	面向网络的服务，包括授权网络、当前网络、信号级别以及搜索模式等各类信息访问
设备制造商（Original Equipment Manufacturer, OEM）功能服务集	允许制造商为电话管理添加功能，并为设备提供新的功能服务集
电话簿功能服务集	访问用户身份模块（Subscriber Identity Module, SIM）和地址簿
电源功能服务集	关于电源供应级别的相关功能
安全功能服务集	为电话和 SIM 卡安全相关功能提供 PIN（Personal Identification Number, 个人身份号码，在这里专指相关密码）管理和相关服务
短信服务功能服务集	启用短信的读取、发送和删除功能
声音功能服务集	手机声音管理，包括按键音和静音的处理
语音通话功能服务集	处理语音呼叫的发送和接收，还包括双音多频（Dual-Tone Multi-Frequency, DTMF）信号的处理

20.9 编程环境

按照有关设计用来运行 Palm 操作系统的计算机上的可用资源，通常无法支撑相关软件的开发。为此，Palm 操作系统编程网站建议，基于 Palm 系统所设计的程序只能支持最低数量的数据项。这一建议在一定程度上是考虑到手写识别输入的难度，同时也是因为非常有限的屏幕显示尺寸。另外，Palm 公司建议有关用户应当主要在桌面系统上输入数据，并利用 Palm 系统来引用相关数据。还有，一旦一个程序运行在了 Palm 操作系统上，往往就不会有简单方便的途径来支持实现任何调试信息的显示，显然这样的环境并不适合程序源代码的输入和编辑。进一步说，几乎没有打印机会连接到 Palm 系统上，而且正如之前所提到的，一般情况下，其处理器运行速度很慢，随机存取存储器空间也很有限，并且往往不会配备辅助存储器。因此，大多数的程序开发都是在另一个系统上完成的，称为跨平台开发。

[475]

有各种语言和工具可用来支持 Palm 操作系统的软件开发。其中，有一些是由 Palm 公司自身提供的，而另外一些则是由第三方提供的，譬如由 Metrowerks 公司提供的 Code-Warrior 等商用的集成开发环境（Integrated Development Environment, IDE）以及一些免费的工具，例如 PRC 工具——一种基于 gcc 的采用 C/C++ 语言来构建 Palm 操作系统应用程序的编译器工具链。由 Palm 公司提供的工具如 Software Development Kit（软件开发工

具, SDK), 具体包括头文件、函数库以及一些在 Windows、Linux 和 Mac 操作系统上用作 Palm 操作系统开发平台的工具。同时, 该软件开发工具还包括一个以原生 x-86 代码运行在 Windows 系统设备上的 Palm 操作系统的版本。这种模拟机制提供了一种简单的方法来测试面向 Palm 操作系统而开发的应用程序的兼容性。除此之外, 还有一些编译器用来支持利用包括 VB、Pascal、Forth、Smalltalk 和 Java 等在内的其他语言来开发应用程序。

被 Palm 公司称为 Emulator (仿真器) 的软件包拥有非常重要的功能。该软件包可以在 Windows、Linux 或者 Mac 操作系统的计算机上仿真 Palm 操作系统平台上的各种型号的硬件设备。鉴于不同的平台往往在其只读存储器内部提供不同的功能, 所以只读存储器镜像就被 Palm 操作系统仿真器用来仿真每种所期望的硬件设备型号。

一旦通过跨平台工具开发完成了一道程序, 该程序就可以通过针对 Palm 个人数字助理而为各种交叉开发平台提供的同步工具安装到 Palm 设备上。

20.10 类似系统和当前发展状况

计算机科学方面的作者所面临的一项困难是, 有关行业的发展变化非常迅猛, 一本书往往无法反映行业的最新发展, 哪怕只是到其印刷出版的那一天的进展情况。操作系统也不例外。Palm 操作系统以及在这一章所提及的其他同类系统所使用的硬件系统已经发生了日新月异的飞速发展。此外, 一种不同的功能视图已经俘获了大众用户和供应商的想法, 并驱使有关操作系统进行某些调整和改变。于是, 按照这种不同视图开发的其他操作系统便拥有了比这里所描述的 Palm 操作系统更为复杂的某些功能。当然, 后来的 Palm 操作系统版本也跟着增加了相应的功能特征。

在本节中, 我们将会讨论面向小微系统的其他操作系统中可以见到的一些功能。我们通常会提到塞班操作系统 (Symbian OS)。该操作系统由塞班公司 (Symbian Ltd) 所开发, 是佩森公司 (Psion) 的 EPOC 操作系统 (译者注: EPOC 一词源自于 “a new epoch of personal convenience”, 即个人便利设备的新纪元) 的衍生版, 只能运行在 ARM (译者注: Advanced RISC Machines 或 Acorn RISC Machine, 即先进的精简指令集计算机机器, 而 Acorn 则是 ARM 公司的前身) 处理器上。同时, 塞班公司是一家手机制造商的联合企业。

20.10.1 新的功能模型

对于小型手持式系统来说, 不仅采用了更为先进的处理器, 而且基本功能也在过去的几年里不断地演化。21 世纪初, 在这一领域的产品还只是大多被想像为个人数字助理或手机。个人数字助理中的应用程序基本上就是电话、地址簿、约会日历、计算器、备忘录、待办事项、专业数据库以及偶尔的专业应用程序。而手机中的主要应用程序则是电话簿 (phone book) 或通讯录 (contact list)。无论哪种情况, 相关应用程序都是独立的, 当然, 偶尔也需要连接到另一台计算机来进行同步、备份或者是加载新的应用程序。在手机中, 真正的电话应用程序是一个实时的任务, 被认为是有关设备的基本功能, 而不是一个单独的应用程序。从额外应用程序的安装并非有关设计方案组成部分的角度来说, 这些手机都是封闭式系统。

然而, 近来, 针对手持式设备的一种新的模型已经演化形成。这种演变发生的原因在一定程度上是由于通信技术可用性的革命以及无所不在连通可用的巨大变革。相关设备现在已被定位为移动通信平台, 但不再仅仅是手机的替代品。将手机和个人数字助理区别开来的主要依据就是: 当个人数字助理打开后, 往往只是使用一些单一的功能, 然后就将其关闭; 而

手机则通常在大多数时间都不是通话状态（即接通状态），但却一直处于连接网络状态并等待来电。除了等待来电，手机也会完成管理连接等其他工作，譬如保持时间同步、与蜂窝网络（cellular network）交互以便于确定当手机打开时其“身”处何方。在手机处理网络连接的同时，为了能够使用个人数字助理的功能，这些移动通信设备的操作系统必须可以在同一时间包含多个活动的任务。在一个纯粹的个人数字助理设备（如最初的 Palm 产品）中，操作系统一般只会提供几个独立的任务，以便让手写识别和同步功能可以在用户界面（User Interface, UI）应用程序运行时使用（或许你还记得由于屏幕尺寸过小，故而没有足够的空间来执行和摆放多个用户界面应用程序）。尽管如此，其并没有为应用程序提供任何单独的后台功能，比如蜂窝网络连接管理和检查来电信息。因此，有关操作系统可以添加更多的功能来支持多线程应用。更为重要的是，除了独立的前台用户界面线程，应用程序现在可以启动线程，并作为后台任务的一部分。利用这一功能的例证就是建立一项服务同时处理多个传输控制协议/网际协议的连接，从而让多个基于传输控制协议/网际协议的应用程序都能够利用单一的传输控制协议/网际协议多线程的服务在同一时间运行。这样，有关传输控制协议/网际协议的服务将作为一个“用户”应用程序而不是作为操作系统内核的一部分来运行。另一个重要的例子就是万维网浏览器。当从服务器获取一个页面时，相关图像和其他被包含的各项内容并不会被自动发送。有关浏览器必须解析初始页面，然后逐个单独请求每一项被引用元素。同时，在从服务器获取其他元素和内容的同时，相应浏览器必须能够继续工作和为用户显示主页面。这样就给用户提供了内容的即时访问和更加流畅的浏览体验。

477

当手机处于接通状态的情况下，其他无须用户界面控制的应用程序则可以受益于这些后台任务。一些较为明显的后台任务，如播放音频文件、下载新的音频文件来进行播放、即时通信（Instant Messaging, IM）以及电子邮件程序连接到服务器来查收新的电子邮件。同时也存在着一些其他的不太明显的后台任务，譬如分布式数据库的同步更新以及所安装软件的升级更新。这些新的功能将有助于手机制造商将它们的产品彼此区分开来。此外，由于用户开始重视通过短信、电子邮件和万维网来即时访问信息，所以他们正在要求这些功能服务。为了提供这些功能，相关设备采用了先进的硬件组件来处理多个通信流、多媒体流，等等。为此，操作系统也必须加以改进和完善。进一步说，应用程序不仅可以启动多个线程，并且可以为每个线程设置不同的优先级，同时还应当设立新的机制来实现在线程之间以及和其他进程之间的同步、与其他进程的内存分段的共享、与其他进程的通信以及数据库的保护。

20.10.2 高级通信模型

就高级通信模型而言，有关操作系统提供了包括加密等安全机制、新的数据链路层模块（譬如蓝牙和 802.11x）以及对应的应用程序接口等其他先进通信机制，以方便新的用户应用程序能够很容易地访问相应的操作系统功能。同时，伴随通信服务中带宽成本的持续下降，我们开始看到越来越密集的多媒体应用程序。根据市场短期预测，流媒体（streaming multimedia）应用将会日益增多，且相关应用程序往往具有大量的硬实时和软实时要求。而随着需求的不断增加，这些小型的操作系统将继续演化和发展。

20.10.3 线程调度

如前所述，手机的使用与个人数字助理的使用有些不同。在一般情况下，个人数字助理不使用的时候就处于关闭状态，但通常情况下手机则一直保持开机运行，从而使其可以等待

来电，并保证有关网络能够不断定位和更新对应手机的位置。另外，蜂窝技术也正是要求这样的实时任务来为网络提供服务的。因此，塞班操作系统中所使用的调度程序就是一个基于优先级的多线程调度程序，任何应用程序都可以是一个“服务器”，并且可以在其地址空间内创建多个执行线程。类似地，鉴于涂鸦式手写识别程序的需要，所以 Palm 操作系统也包括一个实时调度程序，不过其用户应用程序并不能创建实时任务或线程。我们曾经在第 8 章讨论过这样的调度程序。

20.10.4 用户界面参考设计

大多数个人计算机的用户界面非常灵活，相关窗口不仅可以填满整个屏幕，而且也可以缩为一个较小的尺寸，或者是到处移动、相互覆盖，等等，具体取决于用户的想法。但是对于这些较小的系统来说，则往往具有比个人计算机更为简单的界面。通常情况下，有关应用程序假定自己的窗口会填满整个屏幕。在对应设备上，通常拥有三种不同类型的用户界面，大致分别代表了手机、个人数字助理和手持式计算机 (handheld computer，或称为掌上电脑) 三种类型。用户界面类 (user interface class) 是每种设备系列共同拥有的特征的抽象。关于相关用户界面类的特性总结如表 20-4 所示。

478

表 20-4 小微系统设备系列

手机	小尺寸屏幕 (垂直方向) 带有数字按键和几个按钮的键盘 几乎没有用户输入 应用程序拥有整个屏幕
个人数字助理	大尺寸屏幕 (垂直方向) 手写笔输入和几个按钮 有限的用户数据输入 应用程序拥有整个屏幕
高级设备	大尺寸屏幕 (垂直方向和水平方向可以转换) (通常是) 完整的标准传统键盘 更广泛的用户输入 应用程序窗口可以相互重叠等

对于操作系统设计人员和应用程序设计人员而言，所面临的挑战在于，所编写的操作系统和应用程序可以运行在上述任何一种平台上，且无须进行重大的改动。对这种移植性的期望迫使系统严格地按照面向对象设计来加以实现，以便于将用户界面功能与应用程序的其他部分相互隔离。据业内人士估计，一个应用程序大约有 80% 的部分可以与用户界面分离开来。

互联网的日益普及导致了相关新型设备中若干标准应用程序的建立。特别地，用户希望发送即时消息、处理电子邮件、欣赏 (观看或聆听) 流媒体信息、浏览万维网、上传多媒体文件到他们的网站。鉴于这些小微系统的屏幕尺寸非常有限，所以网页通常也就设置为至少 800×600 像素。因此，一部手机上的浏览器必须努力做到可以智能地显示一个更大尺寸的网页。为了构建网页以便在手持式系统上加以查看，最初还曾创立了一个单独的标准——无线标记语言 (Wireless Markup Language, WML)，是无线访问协议 (Wireless Access Protocol, WAP) 的组成部分。然而，后来的发展似乎表明，标准的浏览器经过修改，就可以支持在手持式系统上显示标准的万维网网页。不言而喻，这是一个非常活跃的研究领域。

另外还创立了一个其他的协议，用于在不合适打电话情况下的简短文本消息的发送。例

如，当接收者处于一个非常嘈杂的环境中（在演讲、艺术表演或会议上）的时候，就可能需要此类协议的支持。同时，信息技术在无须复杂交互的场合也可以被用来实施短小的查询。这种协议就是短信服务（Short Message Service, SMS）协议，其支持发送长达 160 字节的消息。短信服务可以像即时通信服务一样用在普通的个人计算机中。不过，即时通信通常情况下是交互性的，而短信服务则往往采用单向消息方式。

20.10.5 位置感知类应用程序

这些系统的另外一个显而易见但却依然值得关注的方面就是其跟着用户四处移动。相关情况在经过一段时间之后逐渐变得明朗化，也就是说，如果有关应用系统能够获悉手机的位置，那么就可以构建一些很有意思的应用程序，称之为位置感知类应用程序。最初的动力可能是已经授权给了手机的紧急定位系统（emergency location system）。在紧急情况下，如果手机可以告知紧急呼叫处理部门及相关人员相应手机所处位置附近几十米的范围，显然是大有益处的。

479

另外也可以构建许多其他的基于定位的应用程序（location-based application）。鉴于在美国的手机运营商已被授权可以使用网络来定位手机的位置，于是他们便决定从此项授权里面来榨取商业价值，进一步说，基于电话当前所处的位置通过设计相关服务并提供给他们的用户（当然，要收取适当的费用）。比如你想知道：最近的饺子馆在什么地方？（译者注：原著此处为 pizza restaurant，但 pizza 即比萨，属于意大利餐饮，和后面的中国餐馆不一致，故做此修改。）于是拨打 1411（或者一些类似的特定的电话号码），接下来，一个态度友好的接线员将会从网络中获取你的位置，询问你需要什么帮助，进而从基于位置索引的数据库中搜索最近的中国餐馆的位置，并将相关信息提供给你。其他常见的应用还比如：一般的驾驶向导、交通报告、天气预报，等等。当然，这些系统也可以完全是计算机自动化的，即无须设立专门的操作人员。

其间存在的一个很有意思的问题是，电话（或网络）如何能够找到手机的当前位置？一种答案是利用联邦的全球定位系统（Global Position System, GPS），其在轨道上运营有几十颗卫星专门用来实现这项功能。最初，它们是为了军事目的而部署的，但是，既然这些卫星的使用仅仅涉及监听它们的广播，故而就不可能永远把民用排除在外。通过几颗卫星的同时定位，任何设备都可以确定其当前的位置，包括海拔高度。这是一种极其精确的定位机制，在许多情况下，只有几英尺的误差（译者注：1 英尺 = 30.48 厘米）。尽管如此，有关硬件成本相比于手机大部分其余部件仍然有些高。幸运的是，另外还有至少两种方法也可以找到电话的位置。其一是通过网络对手机进行三角定位。所有能够捕捉到手机信号的蜂窝基站都会报告来自手机的信号的时间，于是有关网络就可以由此实现手机定位，误差在一百英尺左右的范围内。对于汽车驾驶来说，这种精度是不够的，但是通常情况下这足以找到比如说最近的邮局。另一种网络定位机制则仅仅报告为移动设备提供服务的那一个蜂窝基站的信息以及可能基于信号传播时间计算得到的手机当前位置到对应信号塔的距离。虽然这种方法相比于三角定位而言更不准确，但对于许多应用场景来说仍然足够精确和可以利用。

20.10.6 后来的 Palm 操作系统版本

Palm 操作系统从第 5 版开始支持 ARM 处理器（ARM processor），而不再是以前平台所使用的摩托罗拉的处理器。同时，其从 5.4 版开始的个人应用集成（Personal Application

Integration, PAI) 软件包被称为 Garnet。PalmSource 公司先是从 Palm 公司分离出来, 后又 Access 有限公司 (Access Co. Ltd.) 收购。他们构建了一版用在移动平台上的支持 Garnet 应用程序接口的 Linux 操作系统。Palm 操作系统的最新版本是第 6 版, 称为 Cobalt。

20.11 小结

在这一章中, 我们就一种简单的现代操作系统——由 Palm 公司开发的 Palm 操作系统——的功能特征和设计机理展开了进一步的讨论。该操作系统是针对小型手持式设备而开发的。虽然它是一种单用户系统, 但是它可以同时运行若干操作系统进程以及少量的应用程序。我们从有关操作系统的进程调度和内存管理功能出发来开启本章的内容。然后, 我们讨论了 Palm 操作系统中若干附加的输入/输出子系统、图形化用户接口编程和网络编程, 并就利用较大型系统中的模拟器和交叉编译器为相关受限环境开发程序的过程进行了解释说明。随后, 我们讨论了面向受限环境的一些类似的操作系统以及它们与 Palm 操作系统之间的不同, 也包括后来的 Palm 操作系统版本, 但实际上是一种运行在另一不同处理器上的不同的操作系统。另外, 我们还讨论了伴随个人数字助理和手机平台的融合而出现的一些新的应用程序类型。

480

参考文献

- Exploring Palm OS: Palm OS File Formats, Document Number 3120-002. Sunnyvale, CA: PalmSource, Inc., 2004.
- Exploring Palm OS: System Management, Document Number 3110-002. Sunnyvale, CA: PalmSource, Inc., 2004.
- Palm OS Programmer's Companion, Volume 1, Document Number 3120-002. Sunnyvale, CA: Palm, Inc., 2001.
- Palm OS Programmer's Companion, Volume 2, Communications, Document Number 3005-002. Sunnyvale, CA: Palm, Inc., 2001.
- Palm OS® Programmer's API Reference, Document Number 3003-004. Sunnyvale, CA: Palm, Inc., 2001.
- Rhodes, N., and McKeehan, J. *Palm Programming: The Developer's Guide*. Sebastopol, CA: O'Reilly & Associates, Inc., 2000.
- SONY Clié, Personal Entertainment Organizer, Sony Corporation, 2001.

网上资源

- <http://www.accessdevnet.com> (Linux平台开发套件)
- <http://www.freescale.com>
- <http://www.freewarepalm.com> (免费的Palm软件)
- <http://www.freesoft.org/CIE/> (链接: 互联网百科全书)
- <http://www.imc.org/pdi/>
- <http://oasis.palm.com/dev/palmos40-docs/memory%20architecture.html>
- <http://www.palm.com> (Palm公司主页)
- <http://www.palmsource.com/developers/>
- http://www.pocketgear.com/en_US/html/index.jsp
- (针对移动设备的软件)
- <http://prc-tools.sourceforge.net> (支持Palm操作系统的编程工具)
- <http://www.symbian.com> (塞班系统)
- <http://www.w3.org/Protocols/> (主要是超文本传输协议)
- http://en.wikipedia.org/wiki/Graffiti_2 (关于Graffiti 2 (第2版涂鸦式输入工具软件) 的文章)
- http://en.wikipedia.org/wiki/Palm_OS (关于Palm操作系统版本演化的历史)

习题

- 20.1 既然几乎没有网站是基于 160×160 像素的屏幕大小的假设而开发的, 那么使用超文本传输协议又是用来支持什么的?
- 20.2 对于 Palm 设备来说, 内存型磁盘驱动程序有什么样的用处?
- 20.3 就本章讨论的手持式平台上所运行的程序来说, 相关程序设计人员如何着手开展它们的创建和测试?
- 20.4 相比于早期版本, Palm 平台增加了哪些新的设备类型和功能? 它们又推动了什么类型的应

用呢?

20.5 vCard 是什么?

20.6 什么是“位置感知类应用程序”?

20.7 请描述三种不同类型的手持式系统系列。

20.8 手机和个人数字助理之间的一项主要区别是:个人数字助理往往是使用时开机、不用时关机,而手机则通常大部分时间都处于开机运行状态。基于这一区别,针对手机而设计的相关操作系统中加入了什么主要功能呢?

481
482

计算机系统总览和体系结构概念

在本附录中，我们将概要介绍计算机体系结构的相关概念，重点是那些与操作系统特别相关的概念。有些读者已经选修过计算机组织或计算机体系结构方面的课程，故而会对这些概念比较熟悉。在这种情况下，附录提供的材料可作为复习使用。对于那些以往没有学习过此类主题的课程的学生，则建议仔细地研读附录内容，因为许多操作系统概念的讨论都是在底层的计算机体系结构的基础上展开的。在操作系统相关概念及设计机理的整个陈述过程中常常会用到这里给出的概念。

我们首先在第 A.1 节中描述了典型计算机系统的主要部件，并讨论了每种部件所承担和执行的函数。然后，在第 A.2 节中，我们讨论了中央处理器及其控制原理。接下来，在第 A.3 节概述了内存和存储器体系的设计理念，在第 A.4 节描述了输入/输出系统的基本概念，在第 A.5 节简要讨论了网络在现代计算中的作用和特点。在此基础上，我们在第 A.6 节给出了一幅关于典型计算机系统部件的更详细的结构图。最后，在第 A.7 节对全部内容进行了总结。

483

A.1 典型计算机系统部件

根据功能和预期用途，计算机系统彼此之间的差别往往很大。从这个角度来说，它们可划分为以下几种类型的系统：

- **个人 (personal)** 台式计算机和笔记本计算机 (notebook computer, 或称为笔记本电脑)，通常情况下每次只能由单个用户所使用。
- **大型服务器 (server)** 计算机，每天为成百上千甚至成千上万的用户提供服务。它们包括存储万维网文档的互联网万维网服务器 (Web server, 或称为网站服务器)、存储大型数据库的数据库服务器 (database server)、为计算机网络存储和管理文件的文件服务器 (file server) 以及运行有关提供了远程访问服务的特定应用程序的应用程序服务器 (application server)。
- **嵌入式 (embedded)** 计算机系统，用在汽车、飞机、电话、计算器、家用电器、媒体播放器、游戏机、计算机网络部件以及许多其他这样的设备中。随着处理器芯片变得越来越便宜，我们将会越来越多的地方看到它们。在将来，我们甚至应当会在今天很难想象的地方看到它们。
- **穿戴式移动 (mobile)** 设备、手机 (cell phone, 或称为蜂窝电话) 以及用于保存约会日历、电子邮件、电话簿和其他信息的个人数字助理 (Personal Digital Assistant, PDA)。今天，这类系统正变得越来越强大，故而难以与个人计算机 (Personal Computer, PC) 区分开来。

为此，很难确定典型的计算机系统应当是什么样子。尽管如此，传统上一致认为，大多数的计算机系统应具备三类主要部件，如图 A-1 所示，也就是处理器 (processor) 或中央处

理器、存储器部件 (memory unit) 和输入 / 输出部件[⊖]。除了这三种主要部件之外, 网络设备也很重要, 其将计算机系统连接在一起, 并支持信息和程序的共享。接下来, 让我们分别就每类部件的主要功能逐一加以简要的描述。

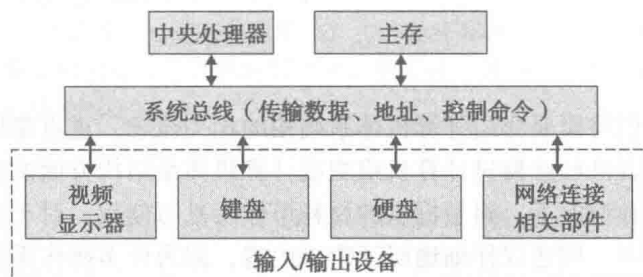


图 A-1 计算机系统结构简图

中央处理器 (Central Processing Unit, CPU, 或称为中央处理单元, 或简称为处理器) 是执行计算机系统所需的计算和控制逻辑的电路。**存储器** (memory) 是用来存储计算所需的数据和有关执行计算的实际控制命令的部件。存储器通常被组织成若干级别, 从而形成关于不同类型存储设备的**存储体系** (storage hierarchy, 或称为存储器层次结构), 我们将在第 A.3 节中详细描述。**输入 / 输出** (Input/Output, I/O) 部件包括基于其主要功能 (即输入和输出) 的两个宽泛的设备子类。当然, 某些设备既可用于输入, 也可用于输出。**输入设备** (input device) 用于从诸如只读光盘 (CD-ROM) 或磁盘之类的设备将数据和程序指令加载到存储器部件中。它们还用于通过诸如键盘或指针式设备 (例如, 鼠标或触摸板) 之类的设备来处理来自用户的输入命令。**输出设备** (output device) 用于通过诸如打印机或视频监视器 (video monitor) 之类的设备为用户打印 (或向用户显示) 数据和信息, 以及在诸如各种类型磁盘的辅助存储器 (secondary storage) 设备上存储数据和程序。需要指出的是, 诸如硬盘驱动器 (hard disk drive) 和读写式光盘 (CD-RW) 驱动器之类的设备既可用于输入, 又可用于输出, 故而被归为**输入 / 输出设备** (input/output device)。网络设备也可以被认为是输入 / 输出设备, 不过其非常特别, 最好还是被认为是单独的某类设备。

如果我们考虑计算机系统的低级硬件视图, 那么一般的磁盘设备 (硬盘、软盘、光盘等) 都应被认为是输入 / 输出设备。而如果我们采取更概念化的视图来看待它们所承担的作用, 即存储数据和程序, 那么它们也可以被认为是计算机系统的存储体系的组成部分, 我们将在第 A.3 节中加以讨论。

在许多现代的计算机系统中, **网络** (network) 是另一类的关键部件, 其是支持现有的数百万台计算机和网络设备实现彼此通信的硬件和软件。网络可以由电话线、光纤和其他类型的电缆, 以及卫星、无线集线器、红外设备和其他部件而形成。但是, 在单台机器级别, 将网络视为另一种类型的输入 / 输出设备有时候却是非常有用的, 因为其主要功能就是从一台 (作为输出的) 机器传送数据 (诸如文件、文本、图片、命令等) 到另一台 (作为输入的) 机器。对于计算机用户来说, 互联网是网络的最明显的例子。

接下来的三节将更为详细地依次讨论处理器、存储器和输入 / 输出三种主要的计算机系

⊖ 在更详细的层面上, 处理器有时被划分为两个部分: 控制器 (control unit, 或称为控制单元) 和数据通路单元 (data path unit), 我们将在下一节中具体展开讨论。类似地, 输入 / 输出部件有时被分成输入设备和输出设备。

统部件。而网络则放在第 A.5 节中加以讨论。

A.2 中央处理器

如前所述,中央处理器是执行各种算术和逻辑运算的硬件电路。每个处理器应当具有一套定义了可由相应处理器所执行的操作的特定的指令集(instruction set)。其中通常包括整数算术运算、比较操作、传送操作、控制操作等。同时,处理器往往会拥有一组寄存器,用来保存正在执行的操作以及对应操作所需的一些数据值或操作数(operand)^①。根据有关指令集的设计,也可以直接从存储器单元(memory location)访问其他的操作数。我们在本节后面的部分将进一步详细说明寄存器的使用以及操作数的类型。

各种指令集千差万别。有些处理器是基于精简指令集计算机(Reduced Instruction Set Computer, RISC)理念而设计的,仅仅在硬件中直接实现了其中的几种基本指令类型,因为相关指令彼此在设计方面往往非常类似。精简指令集计算机的优点之一是通过拥有一个包含有限指令类型的集合来降低硬件的复杂性,并因而提高相关指令的执行速度。最常见的精简指令集计算机微处理器是 Alpha 系列(由惠普公司开发,尽管不再制造,但在历史上具有重大意义)、嵌入式 ARM 处理器、MIPS、PIC 微控制器系列、PowerPC 系列(由苹果公司、IBM 公司和摩托罗拉联合开发)以及 SPARC 系列(由太阳微系统公司开发)。

其他处理器则拥有一套直接在硬件中实现的大得多的指令集,其中包含了各种各样的指令类型。这种方法被称为复杂指令集计算机(Complex Instruction Set Computer, CISC)。精简指令集计算机处理器通常具有 30~100 条具有 32 位固定指令格式的不同指令,而复杂指令集计算机处理器则往往拥有 120~400 条不同的指令。复杂指令集计算机处理器的例子如 IBM 公司的 System/360、DEC 公司的 VAX 和 PDP-11、摩托罗拉的 68000 系列以及英特尔的基于 x86 架构的处理器和兼容处理器。

今天的大多数处理器既不是纯粹的精简指令集计算机,也不是纯粹的复杂指令集计算机。这两种设计理念实际上都在朝彼此方向演化发展,故而相关方法在提高性能和效率方面不再具有明确的区别。采用精简指令集计算机的各种指令集的芯片增加了更多的指令和复杂性,因此现在它们与复杂指令集计算机的同类产品一样复杂,而相关争论主要发生在有关的营销部门之间。

A.2.1 指令集体系结构及机器语言

指令集体系结构(instruction set architecture)定义了处理器的机器语言(machine language),而机器语言就是处理器可以直接执行的命令的集合。每条指令被编码为可以由处理器解码和执行的二进制位序列,即位串(bit string)。指令存放在存储器中,并且通常情况下按顺序执行,除非当某些类型的指令指定了特定的控制转移操作(transfer of control)的情况下。指令位串被划分成若干部分,称为字段(field)。尽管各种指令格式相互之间的区别可能很大,但是往往都包含有一些典型的字段,具体说明如下:

- 操作码(operation code, opcode)字段,用来指定要执行的特定操作。
- 修饰符字段(modifier field),有时被用来区分具有相同操作码和格式的不同操作,

① 至少在概念上,存储数据值的寄存器也可以被认为是存储器层次结构的顶层(参见第 A.3 节),因为它们用来保存数据并且在被所执行的指令访问时提供了最快的访问速度。当然,从物理上讲,它们是处理器芯片的组成部分。

例如整数加法和减法。

- **操作数字段 (operand field)**, 用于指定每个特定操作所需的数据值或地址。有关地址可以是存储器地址, 也可以是寄存器地址。
- 有许多不同类型的操作数, 而用来解释每种类型的操作数的含义的方式称为**寻址方式 (addressing mode)**。我们可以区分两种主要类型的操作数。第一种类型提供了操作所需的数据值 (data value) 或数据值的地址 (address of a data value)。第二种类型提供了指令的地址 (address of an instruction, 或称为指令地址), 其一般在通过分支操作 (branch operation) 或跳转操作 (jump operation) 来改变指令执行顺序的情况下才会使用。

关于**数据 (data)** 操作数, 最常见的几种寻址方式说明如下:

- **寄存器寻址 (register addressing)**: 对应操作数指定了用来存储有关操作所需的或所生成的数据的寄存器的地址。
- **立即寻址 (immediate addressing)**: 对应操作数就是包含在相应指令位串本身某一字段中的直接的数据值。
- **基址寄存器寻址 (base register addressing)**: 对应操作数存放在存储器单元中, 并且该存储器单元的地址应通过将**基址寄存器 (base register)** (其包含有一个用作参照的存储器单元的地址) 的内容和**位移量 (displacement)** 或**偏移量 (offset)** 相加来计算获得。有关位移量可以是指令本身中所包含的直接数值, 也可以是另一个寄存器中所包含的数值, 在后一种情况下把相应的另一寄存器称为**索引寄存器 (index register)**。
- **间接寻址 (indirect addressing)**: 如果要用作操作数的数据的存储器地址被存放在一个寄存器或另一个存储器单元中, 则称之为间接寻址。因为有关指令并没有指向要在操作中所使用的数据, 而是指向了存储器中或某个寄存器中的相关数据 (即实际数据地址) 的地址, 并且该地址必须首先被访问才可以获得所需的实际数据地址。

关于**指令地址 (instruction address)** 操作数, 最常见的两种寻址方式说明如下:

- **程序计数器相对寻址 (PC-relative addressing)**: 有关指令的存储器地址可以通过把程序计数器 (Program Counter, PC) 寄存器的内容与一个偏移量相加来计算获得。通常情况下, 程序计数器寄存器保存着要执行的下一条指令的地址。于是, 与基址寄存器寻址方式一样, 相关偏移量可以是包含在指令本身中的直接数值, 也可以是某个索引寄存器中的数值。
- **间接寻址 (indirect addressing)**: 有关指令的存储器地址存放在一个寄存器或另一个存储器单元中。与数据操作数的间接寻址方式一样, 有关指令并没有指向要转移的地址, 而是指向了存储器中或寄存器中的相关数据 (即实际转移地址) 的地址, 并且必须首先访问该地址才可以获得所需的实际转移地址。

487

部分上述寻址方式如图 A-2 所示。

一般情况下, 操作类型将会决定如何来解析操作数——无论是作为存储器地址, 还是作为指令地址, 亦或作为直接数据值, 也可能是以某种其他方式。通常情况下, 精简指令集计算机处理器拥有的寻址方式较为有限, 而复杂指令集计算机处理器则往往拥有种类繁多的寻址方式。

在这里，我们举例说明一些简单的指令格式以及相应的寻址方式，如图 A-2 所示。

图 A-2a 具体说明了一种加法 (add) 运算指令，也就是说，把寄存器 A 和寄存器 B 的内容相加，并把结果存放到寄存器 C 中。其间，要加的数值必须事先通过先前的指令分别加载到寄存器 A 和寄存器 B 中。

图 A-2b 具体说明了另一种加法运算指令，其中的一个操作数是存放在指令本身中的立即数据值。该运算指令将寄存器 A 的内容和立即操作数的数值相加，并将结果存放到寄存器 C 中。

图 A-2c 具体说明了一种加载 (load) 操作指令，用于把存储器单元中的数据值放入寄存器 A 中。该指令采用了基址寄存器寻址方式来计算相应的存储器单元地址。具体而言，把基址寄存器中的内容和索引寄存器中的内容相加，对应的和值结果被用作存储器单元的地址，且对应的存储器单元的内容被传送和加载到结果寄存器 A 中。

图 A-2d 具体说明了一种无条件跳转 (jump) 操作指令，用来将控制转移到下一条指令之外的其他某条指令处。其根据基址寄存器的内容和立即数据值来计算下一条要执行的指令的存储器单元地址。进一步说，通过将基址寄存器的内容与存放在指令本身中的立即操作数 (作为索引或偏移量) 相加，就可以得到下一条即将执行的指令的存储器单元地址。

图 A-2e 具体说明了一种相等条件下的分支跳转 (conditional branch-on-equal) 操作指令。该指令首先比较寄存器 A 和寄存器 B 的取值。如果二者的取值相等，那么就将控制转移到这样的一条指令上，即所转去执行的指令的存储器单元地址是根据程序计数器寄存器的内容 (下一条指令的地址) 与当前指令中的立即数据值相加所得到的。这样的指令通常可以用于循环控制。

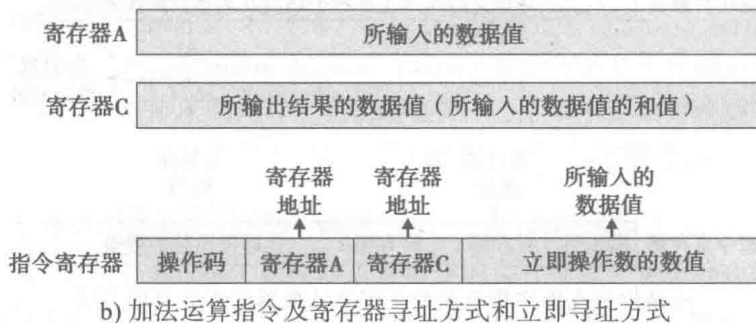
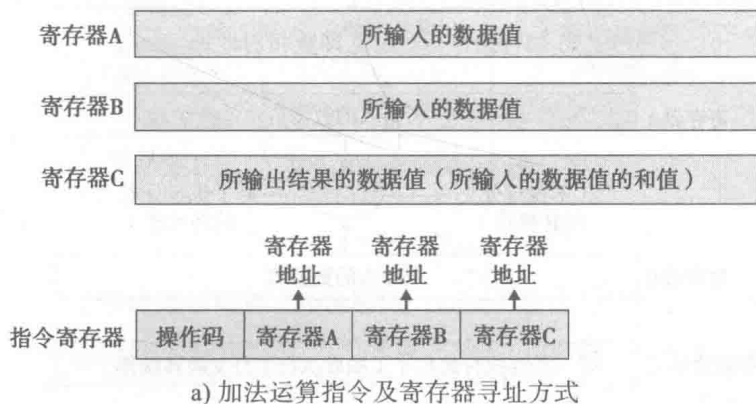
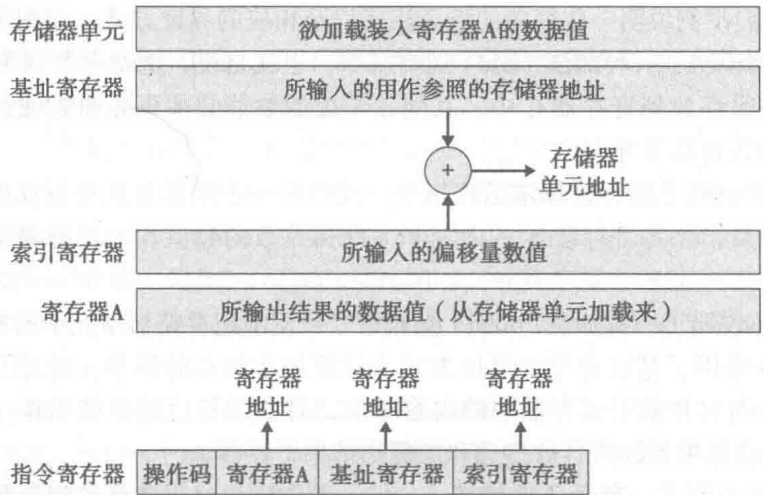
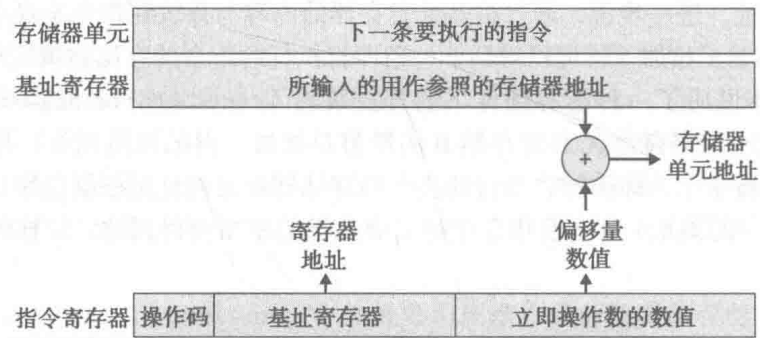


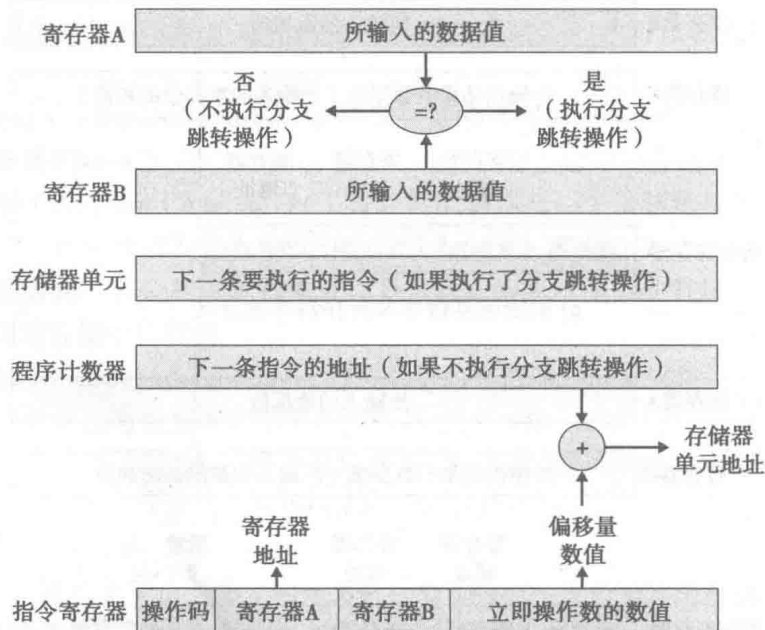
图 A-2 寻址方式和指令格式



c) 加载操作指令及寄存器寻址方式 (涉及基址寄存器和索引寄存器)



d) 跳转操作指令及立即数相对地址



e) 条件分支操作指令及基于程序计数器的索引寻址方式

图 A-2 (续)

除了确定不同的寻址方式——确定如何解释操作数的位置和取值——之外，许多处理器还具有两种执行模式（execution mode）。当用户或应用程序执行的时候，一般使用的是用户模式（user mode）。而当操作系统内核例程执行的时候，则往往使用监管模式（supervisory mode，或称为管态）或特权模式（privileged mode）。处理器中的某个特殊寄存器用来确定系统当前正在使用哪种执行模式。当处于用户模式时，将会在指令执行期间启动实施某些保护机制。例如，在用户模式下将启用内存保护机制，以禁止用户程序去访问有关分配给该程序的内存部分之外的存储器单元。同时，某些特权指令（譬如控制输入/输出设备的指令）只有在系统处于监管模式的情况下才允许执行。

A.2.2 处理器组成部件

图 A-3 是关于处理器典型部件及组成结构的一个简图。其中，整数算术逻辑部件（Arithmetic and Logic Unit, ALU）和浮点部件（floating point unit）包括执行指令集运算操作的硬件电路。大多数常规指令均由整数算术逻辑部件加以处理，而浮点算术指令则由浮点部件负责处理，因为相关运算操作需要更复杂且高度专用的电路。控制器（control unit）通常包括处理器寄存器以及各种控制电路，例如用于控制指令执行顺序的电路、用来解析有关指令代码和操作数的控制电路以及用来控制有关算术逻辑部件或浮点部件电路的指令的执行的电路。

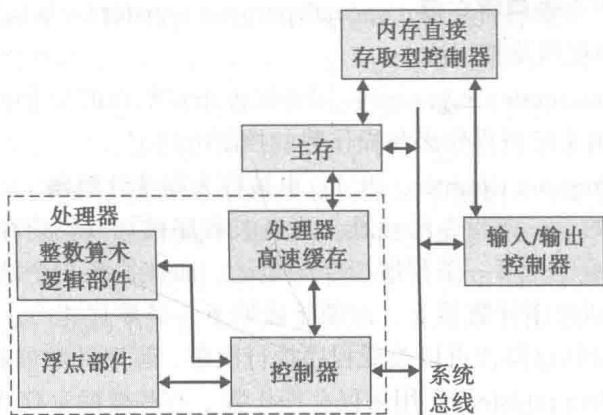


图 A-3 处理器典型部件及结构简图

图 A-3 中所示的处理器高速缓存（processor cache）是作为处理器芯片组成部分的存储器部件，并且保存有当前处理器正在使用的来自主内存（main memory，简称主存或内存）的指令和数据。（另外，在处理器芯片自身外部也可能存在其他的高速缓存存储器。）处理器高速缓存经由单独的存储器总线（memory bus，或称为内存总线）连接到了主存上。同时，主存也与主系统总线（system bus）相连接。而控制器则通过系统总线连接到了输入/输出设备上。内存直接存取（Direct Memory Access, DMA）型控制器是另外的一个部件，支持将数据直接从输入/输出设备传送到内存。我们将在第 A.3.2 节中详细讨论缓存机制及相关概念，而内存直接存取型控制器已在第 14 章进行过讨论，这里不再赘述。

A.2.3 源程序、目标程序和可执行程序

汇编语言（assembly language）是机器语言指令的替代形式，更方便也更容易被人阅读（和编写）。在汇编语言中，每个可能的操作码被赋予了一个助记符（mnemonic name），即一个用来识别相应指令的符号名称。操作数可以呈现为数字形式，或者也可以是用来标识

映射到内存地址或寄存器的程序变量的名称。采用诸如 C++ 等高级程序设计语言编写的程序——称为**源代码** (source code) 或**源程序** (source program)——可以由对应的程序设计语言**编译器** (compiler) 转换为机器语言的程序, 称为**目标代码** (object code) 或**目标程序** (object code program 或 object program)。在此基础上, 此类目标代码程序与其所需的来自程序库和其他程序模块中的目标代码程序相链接, 就可以创建**可执行代码程序** (executable code program, 或 executable program, 即可执行程序) 文件。其通常作为二进制文件存储在磁盘上, 故而有有时被称为**程序二进制** (program binary, bin) 文件。当需要时, 可执行代码将被加载到内存中, 然后, 有关程序指令就会在处理器上按照期望的顺序加以执行。

[491]

良好的程序设计语言编译器应当在创建目标代码时充分利用对应机器指令集中的各种指令。因此, 开发编译器的程序设计人员必须认真仔细地研究每台机器的指令集体系结构, 以便于充分利用相应的功能。

A.2.4 处理器寄存器、数据通路及控制

在处理器的组成部分中有几种类型的寄存器, 它们由处理器电路采用各种方式来加以使用。一些处理器使用了**通用寄存器** (general-purpose register), 其中相同的物理寄存器可以按照下面所讨论的多种方式甚至所有方式进行使用。而在其他处理器的设计中, 则把一些寄存器或者所有寄存器作为**专用寄存器** (special-purpose register), 并且只能用于特定的功能。如下是最常见的几类寄存器及其用法:

- **指令寄存器** (instruction register): 用来保存正在执行的指令。它们直接连接到有关在执行指令时用来解析操作码和操作数的控制电路上。
- **程序计数器** (Program Counter, PC): 也被称为**指令计数器** (instruction counter), 用来保存要执行的下一条指令的地址。当有关程序被加载到内存并被启动执行时, 程序计数器被初始化为第一条程序指令的地址。而在执行指令的时候, 当前指令的长度通常会被加到程序计数器上, 以便于提取下一条顺序指令。当然, 分支或子程序调用或其他控制转移操作可以改变程序执行次序、程序计数器取值以及更新方式。
- **数据寄存器** (data register): 用来保存操作数。有些数据寄存器可专用于保存特定的数据类型的数据, 例如, 浮点寄存器 (floatingpoint register) 只能用来保存浮点操作数。小型处理器往往可能只有一个主数据寄存器 (main data register), 通常称之为**累加器** (accumulator)。在某些情况下, 还会有一个用来保存较大操作数或除法运算余数的附加寄存器, 而且一般称之为**累加器扩展寄存器** (accumulator extension register)。
- **地址寄存器** (address register): 用来保存存储操作数或指令的主存地址的值。它们可以保存绝对存储器地址 (absolute memory address, 或简记作 absolute address, 即绝对地址) 或者相对存储器地址 (relative memory address, 或简记作 relative address, 即相对地址), 后者也就是偏移地址 (offset, 或称为偏移量), 且偏移地址与**基址寄存器** (base register) 的内容相加就可以计算获得绝对地址。保存相对地址的寄存器被称为**索引寄存器** (index register)。
- **中断寄存器** (interrupt register): 用来保存有关可能发生的中断事件的信息, 我们将在稍后加以讨论。
- **程序状态寄存器** (program status register): 用来保存处理器所需的控制信息。不同的机器可能具有任意数量的状态寄存器, 并且内容相差很大。举例说明它们所保存的

各类控制信息如下:

- 最后一次比较操作的结果 (即 $a > b$ 、 $a = b$ 或 $a < b$)
- 处理器状态 (即处理器是处于用户模式还是监管模式)
- 错误状态, 例如算术溢出、除零错误等

492

- **时钟 (clock):** 时钟寄存器 (clock register) 实际上是一种倒计时到零并产生中断——称之为**时钟中断 (clock interrupt)**或**定时器中断 (timer interrupt)**——的定时器 (timer)。鉴于各种原因, 时钟中断可以由操作系统来进行设置。例如, 在多用户系统中, 有关操作系统通常在被称为时间片 (time quantum, 或称为时间量) 的有限时间期限内把处理器控制权交给用户程序。而通过设置定时器中断, 如果有关用户程序在时间片结束时还在运行, 那么操作系统就可以中断该用户程序, 并检查其他程序是否正在等待要运行在相应的处理器上。时钟中断也可用来计算实际的日期和时间。需要强调的是, 处理器常常具有一条特权指令且只能由操作系统执行, 用来将一个时间量数值加载到时钟寄存器中进行时间的设定, 所以用户程序不能绕过操作系统直接推翻或撤销操作系统所设定的时钟值。

当然, 有些寄存器可以由用户程序直接设置, 故而被称为**用户可见寄存器 (user-visible register)**。它们通常包括数据寄存器、地址寄存器和指令寄存器。但其他的寄存器往往只能由处理器或操作系统内核进行设置, 例如状态寄存器和中断寄存器。精简指令集计算机处理器通常拥有大量的通用寄存器, 因为它们具有统一的指令集设计。相比之下, 复杂指令集计算机处理器则常常同时兼有通用寄存器和专用寄存器。某些类型的寄存器使用可能要求专用寄存器, 例如中断寄存器、程序状态寄存器和指令寄存器。

(根据操作码来) 识别特定指令的电路以及利用操作数执行指令的电路与指令寄存器相连接。由于指令执行牵涉通过有关硬件电路从寄存器和内存来传送信息 (操作码、操作数等), 故而这些电路有时被称为处理器的**数据通路 (data path)** 部件。另一方面, 用来控制下一条指令的提取的电路以及用来控制诸如中断 (详见后面描述) 等其他事件的处理的电路被称为处理器的**控制部件**。

A.2.5 系统时序

系统时钟 (system clock) 是各种处理器内部另外的一种用来实施系统时序 (system timing) 的重要部件。大多数逻辑电路的操作都是以同步节拍方式向前推进的。在电子级别, 这被称为系统时钟。(这不应与操作系统用于定时的处理器寄存器混为一谈。) **系统时钟周期 (clock cycle)** 是指期间可能发生处理器操作的固定的最短的时间间隔。处理器的速度取决于系统时钟每秒产生的周期数。一个 10 亿赫兹的处理器每秒将会拥有 10 亿个时钟周期。处理器技术和指令集设计是决定处理器总体速度的主要因素, 且简单指令的执行完成比复杂指令将花费更少的时钟周期数。于是, 这被认为是精简指令集计算机机器的一项优点, 因为精简指令集计算机的指令通常比复杂指令集计算机的指令以更少的时钟周期数来执行完成。

A.2.6 指令执行周期和流水线

通常情况下, 典型的指令执行周期一般可划分为以下 5 个阶段:

- **指令提取 (instruction fetch):** 把有关指令从存储器中取出, 并放到指令寄存器中。
- **解码 (decode):** 分析和译解操作码, 并确定有关输入操作数的位置。

493

- **数据提取 (data fetch)**: 如果需要, 就从存储器中提取相应的操作数。
- **执行 (execute)**: 执行对应的操作。
- **写回结果 (write-back)**: 把操作输出结果存放到适当的位置。

注意, 有关指令或操作数可能是在高速缓存存储器 (cache memory, 或称为高速缓存) 中而不是在主内存 (primary memory, 简称为主存或内存) 中。对于许多简单指令来说, 每个阶段通常需要一个时钟周期, 尽管这可能会根据处理器、指令类型以及操作数寻址方式而有所不同。因此, 一条简单的指令从开始到完成将需要 5 个时钟周期。为了提高程序执行的速度, 大多数现代的处理器的都采用了一种称为**流水线 (pipelining)**的策略, 即允许前后连续指令的执行阶段相互重叠。例如, 当一条指令处于该指令的写回结果阶段时, 下一条指令将处于对应指令的执行阶段, 而再下一条指令将处于相应指令的数据提取阶段, 以此类推。只要所有指令均按照顺序次序加以执行, 故而处理器就可以预先知道它们的执行次序, 那么这种策略就能够发挥效用。在这种情况下, 将可能实现指令处理的 5 倍提速。

流水线处理器 (pipelining processor) 必须包括针对诸如分支和跳转指令等改变执行次序的指令的应对处理措施。进一步说, 一次跳转操作将会终止一个执行流水线 (execution pipeline), 并会在另一不同的指令位置开启另一个执行流水线。而如果在有关指令的执行周期开始之后确定了分支转移操作, 那么已经历了它们的执行周期的某些步骤的指令有可能不得被取消 (撤销)。还有, 如果一条指令需要由其前一条指令产生的操作数作为其输入, 那么有时候就需要延迟相应的流水线。因此, 通过流水线实际达到的提速效果必须通过根据许多不同程序可以获得的提速结果的平均值来进行估算。

A.2.7 中断

中断 (interrupt) 是处理器所包含的一项重要功能。中断与操作系统密切相关, 正如我们在本书中所看到的, 操作系统采取各种各样的方式对中断进行了利用。通常情况下, 中断是一种**异步事件 (asynchronous event)**, 即可能在任何时间发生的事件, 故而与系统时钟和处理器指令执行周期并不同步。中断将向处理器发出信号, 告知其需要处理高优先级事件。处理器硬件往往包含了可由中断事件 (interrupting event) 进行设置的一个或多个**中断寄存器 (interrupt register)**。

每当指令执行完成的时候, 有关控制电路就会自动检查是否有中断事件对中断寄存器的内容进行了某种设置。因此, 只有在指令切换的时候, 而不能在指令执行的过程中来为中断提供服务, 即处理中断事件^①。如果检查确认发生了中断事件, 那么**处理器状态 (processor state)**——包括程序计数器以及在中断处理期间即将被使用的任何寄存器的内容——应当被保存到存储器中, 并且跳转到处理中断的程序代码处继续执行。一旦完成了有关中断处理程序 (interrupt handler), 那么系统通常将恢复处理器状态, 并从当初中断的位置起恢复相应的用户程序的执行。而如果中断导致当前程序被终止或者暂停, 那么操作系统就可以切换到另一程序加以运行。

在处理中断的同时, 往往会禁用那些低优先级或者不太重要的中断, 直到当前中断处理完成, 具体由操作系统通过设置**中断禁用 (interrupt disable)**寄存器或**中断屏蔽 (interrupt mask)**寄存器来予以实现。根据该寄存器中的取值, 系统就不会去检查较低优先中断级别上

① 在采用流水线技术的情况下, 每当一条指令完成其执行周期的时候, 就会检查中断。故而处理器应当提供相应机制来善后处理中断处理时那些部分执行尚未完成就被取消的后续指令。

的中断。因此，操作系统便可以在开始中断处理之前先对该寄存器进行设置，并在完成中断处理之后再对其进行复位。

我们可以把引发中断的事件划分成硬件事件（hardware event）和软件事件（software event）。一般来说，硬件中断（hardware interrupt）是异步的，而软件中断（software interrupt）是同步的。可触发中断的各类典型硬件事件说明如下：

- 发生了用户的一些输入/输出操作，例如鼠标移动或鼠标按钮点击或键盘输入。在这种情况下，相应中断处理程序将检索和取回有关输入/输出操作的信息，譬如鼠标坐标或从键盘输入的字符。
- 完成了磁盘输入/输出传输操作。在这种情况下，相应中断处理程序应检查是否存在被挂起的其他的磁盘输入/输出操作，如果存在，则启动下一项磁盘输入/输出传输操作，把对应数据从磁盘传输到主存中或者从主存传输到磁盘上。
- 发生了时钟定时器中断。在这种情况下，应允许操作系统将处理器分配给另一道程序。

可以引起中断的软件事件可以进一步细分为异常（trap，或称为陷入事件）和系统调用（system call）。前者是指当程序发生错误或违规操作时发生的软件事件，而后者则指当用户程序向操作系统请求提供服务时所发生的软件事件。（由于历史原因，系统调用中断有时也被称为陷入——这其实有点混淆。）导致异常发生的一些事件说明如下：

- 违反存储保护的操作事件，例如，一道以用户模式执行的程序试图访问在其允许的存储器空间之外的存储器区域。
- 违反指令保护的操作事件，例如，一道以用户模式执行的程序试图执行某条保留为监管程序模式才可执行的指令。
- 诸如除以零之类的指令错误。
- 诸如浮点数溢出之类的算术错误。

我们通过这本书详细地讨论了有关操作系统是如何处理以上这些事件以及其他引发中断的事件的。

A.2.8 微程序设计

在一些计算机中，通常采用微程序设计（microprogramming）的概念，而将复杂指令实现为基本指令的序列。换句话说，微程序（microprogram）是指实现较复杂操作的基本操作的序列。相关序列存放在处理器中的特定的微程序存储器中，以便在执行复杂指令时能够加以调用。微程序有时也被称为固件（firmware）。有些处理器体系结构（通常是复杂指令集计算机）采用了微程序设计技术，但其他的处理器体系结构则未予使用。

A.2.9 处理器芯片

历史上，处理器是由诸如继电器、电子管、晶体管或简单集成电路等分立元件而构建的。在现代系统中，整个处理器通常被实现为单一的集成电路（芯片）。处理器芯片（processor chip）通常包含有处理器、时钟、寄存器、高速缓存存储器以及取决于特定处理器设计的其他可能的电路。

A.2.10 多核芯片

在过去的几年中，处理器集成电路的制造商已经得出了这样的一个结论，即对更快的处

理器的需求将会有所减缓。为此，他们已经开始着手尽量利用芯片上的额外空间来封装多个处理器，即多核芯片（multicore chip）。同时，关于高速缓存存储器的放置，也存在各种各样的选择设计方案。我们将会在下一节中就有关高速缓存机制展开讨论。尽管这看起来是一个相当微不足道的改变，但是我们在内存的相关章节中能够看出，高速缓存机制对于操作系统却是比较重要的。目前，具有4个处理器核（CPU core）的芯片已十分常见。预测在未来的几年内，将会朝着最多128个处理器核的方向继续迈进。

诚然，要编写能够同时有效地使用多个处理器的程序确实非常困难。然而，大多数的用户都有许多程序在同时运行，所以让多个处理器来运行这些程序必然意味着它们都将运行得更为快捷。而且，大多数的用户经常使用的也就那么几个程序，并且它们均已经过精心设计开发和做好了使用多个处理器的准备，这些程序包括我们使用最多的那些程序中的大部分——字处理程序、电子表格、浏览器等。

A.3 存储器部件和计算机存储体系

A.3.1 存储单位：位、字节和字

存储器部件（memory unit）是存储处理器所需的有关程序指令和操作数的硬件。基本的物理存储单位是单个的二进制位，或简称位（bit，或称为比特），其存储了一个二进制的零（0）值或一（1）值。在现代系统中，位经常（按每8位一组而）被编组为字节（byte），并且字节又常常被编组成字（word）。通常情况下，每个字为4字节或8字节，但是为嵌入式系统所设计的处理器也可能具有1字节或2字节的字。正常情况下，在存储器部件和处理器之间传送的基本单位是字。而一般来讲，往往设立有允许加载或存储单个字节或半字（half word）的指令。在大多数的系统中，每个字节均具有唯一的存储器地址（memory address，有时也称为内存地址）。给定一个特定的存储器地址，有关存储器电路就能够将该特定字节定位在存储器中的特定位置上，进而就可以将包含该字节的字传送到处理器或者从处理器传送到存储器中的特定位置上。存储器字节或字也可以传输到输入/输出设备，或者从输入/输出设备传输到存储器中。在许多情况下，存储器和输入/输出设备之间直接传送的往往是由多个字所组成的数据块，或简称为块（block）。

字大小（word size）通常是处理器寄存器的标准大小。为此，32位处理器应当具有32位，即4字节的标准数据项。另一方面，16位处理器通常具有与许多旧式的个人计算机处理器一样的16位数据格式。某些处理器则具有64位，即“双字”（double word）的数据大小。曾几何时，这种字大小主要是出现在大型计算机上的。但是现在，大多数的处理器都是32位的，而且，今天的个人计算机正在向64位格式演化。尽管一些操作数也可以是单字节或2字节或8字节，然而，许多操作数的大小经常是一个字的大小（4字节）。特定的操作码将确定每个操作数的类型和大小。

随着基本数据字大小（basic data word size）从16位扩展到64位，指令格式的大小也在变长，主要是希望能够使用更大的存储器地址。复杂指令集计算机机器中的指令往往是可变长度的，因为其仅仅需要几位来指定寄存器，而许多位则被用来指定存储器地址。根据寻址方式，相关指令往往可以指定从0个到3个存储器地址，故而对指令长度也将相应地变化。

A.3.2 存储体系

当前的大多数系统都具有多个级别的存储器，通常称为存储体系（storage hierarchy，或称为存储器层次结构），如图 A-4 所示。关于存储体系的传统视图一般拥有三个级别：主存储器、辅助存储器和第三级存储器。接下来我们分别展开进一步的讨论。

主存储器由主内存（main memory，简称主存或内存）以及通常的一个或多个高速缓存存储器（cache memory，或简称高速缓存）组成，甚至有时候处理器寄存器也被认为是主存存储体系的一部分。为此，在主存储器中，可以有若干级别。如果我们认为处

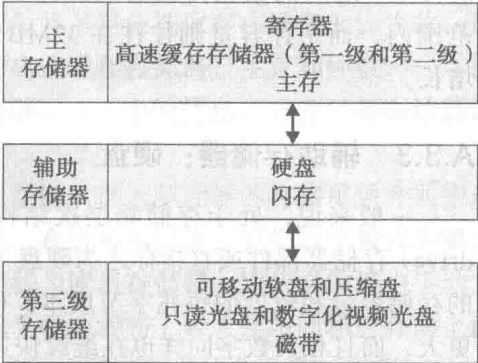


图 A-4 存储层次结构

理器寄存器是内存体系的组成部分，那么它们应当处于最顶层。在下一级别是高速度、低容量的高速缓存存储器，其通常被包括为处理器芯片自身的一部分。在主处理器芯片外部也可能存在附加的高速缓存存储器，但其存取速度要慢于先前级别的高速缓存存储器，而存储容量要大于先前级别的高速缓存存储器。在更低的级别上，是包含在一个或多个单独的芯片上的较低速度但较大容量的主存。高速缓存存储器通常使用被称为静态随机存取存储器（Static Random Access Memory，SRAM）的硬件技术，而主存则往往使用动态随机存取存储器（Dynamic Random Access Memory，DRAM）的技术。与动态随机存取存储器技术相比，静态随机存取存储器技术的每存储单位的存取速度更快，但存储成本更贵[⊖]。

497

处理器寄存器比高速缓存存储器或主存能够更快地读取或写入。例如，在精简指令集计算机处理器中，寄存器到寄存器的复制仅仅需要耗费单个时钟周期即可完成，寄存器到高速缓存存储器的传输需要耗费两个时钟周期，而寄存器到内存的传输则可能需要耗费 3~4 个时钟周期。

高速缓存存储器通常划分为两部分：用于存储操作数的数据高速缓存（data cache）和用于存储指令的指令高速缓存（instruction cache）。在某些情况下，会为在用户模式下的应用程序和在监管模式下的内核分别设立不同的高速缓存部分。高速缓存和处理器之间的字节传输比主存和高速缓存之间的字节传输要快若干倍。因此，有关目标是在高速缓存中保持存放有当前正在使用的数据和指令。这项任务是处理器中的高速缓存管理电路的责任，但是程序设计也可能会影响硬件实施高速缓存所需指令和数据的能力。

内存容量通常按 KB（Kilobytes，即 1024 字节）、MB（Megabytes，即 1 048 576 字节）、GB（Gigabytes，即 1 073 741 824 字节）甚至 TB（Terabytes，即 1 099 511 627 776 字节）来进行度量。由于高速缓存比主存更加昂贵，所以其容量往往要小得多。许多处理器具有两个高速缓存：处理器芯片上的 1 级或 L1 高速缓存（L1 cache），以及处理器外部的 2 级或 L2 高速缓存（L2 cache）。有些处理器甚至还拥有在处理器外部的第三级的 L3 高速缓存（L3 cache）。一般而言，较高级别（1 级最高，2 级次之，3 级再次之，以此类推）的高速缓存比较低级别的高速缓存更快，但是也更昂贵并且保存较少的信息。

⊖ 内存、处理器和磁盘技术总是在不断地发展变化，故而在任何时候都有可能投入使用更新的技术。我们不会就不同类型的存储器是如何在硬件级别上实际构建的主题展开进一步的讨论，因为这与我们所介绍的内容并不直接相关。

内存总线 (memory bus) 是处理主存 (在内存芯片上) 和高速缓存存储器 (在处理器芯片上) 之间的数据传输的硬部件。高速缓存存储器大小通常在 64KB 到若干 MB 之间的范围内, 而主存容量则往往在 32MB~4GB 之间的范围内。不过, 这些数字还在继续迅速增长。

A.3.3 辅助存储器: 硬盘

一般来说, 处于存储器层次结构中的下一级是**磁盘硬盘驱动器** (magnetic disk hard-drive) 存储器部件或直接称之为**硬盘** (hard disk)。硬盘比主存访问速度慢, 但是具有非常大的存储容量和较低的每兆字节成本。硬盘存储容量通常在 10GB~1TB 的范围内, 也可能会更大, 而且相关数字同样也在继续快速增长。硬盘是大多数独立的计算机系统的一部分, 但是通常不包括在诸如个人数字助理、音乐播放器、电话、汽车、家用电器等各种设备上所使用的嵌入式系统中。传统上, 寄存器、高速缓存存储器和主存一起被称为**主存储器** (primary storage), 而硬盘则被称为**辅助存储器** (secondary storage)。每个系统必须拥有一个主存储器部件。

498

主存储器和辅助存储器之间的重要区别是所谓的**存储易失性** (storage volatility)。在**易失性存储器** (volatile memory) 中, 当电源关闭时, 存储器的内容将会丢失。而在**非易失性存储器** (nonvolatile memory) 中, 当电源关闭时, 存储器的内容仍将保持和继续存在。大多数的主存储器系统都是易失性的, 而大多数的辅助存储器系统都是非易失性的。因此, 在由于电源故障而导致系统崩溃的情况下, 磁盘也可用作后备存储器介质^①。

在硬件级别, 主存储器和辅助存储器之间的数据传输牵涉输入/输出设备控制器, 我们将会在第 A.4 节中加以讨论。设备控制器通常具有存储器部件用来保存在磁盘和主存之间所传送的数据。此类存储器部件称为**磁盘高速缓存** (disk cache, 或称为**磁盘缓存**) 或**控制器高速缓存** (controller cache, 或称为**控制器缓存**)。

通常情况下, 因为设备控制器拥有自己的处理器以及自己的不与处理器时钟同步的时钟, 所以往往需要此类高速缓存。一旦处理器启动了传输操作, 处理器将会把对传输的实际控制交给输入/输出控制器, 而后处理器继续执行程序。因此, 这时候主存就被处理器和设备控制器一起访问。鉴于处理器对内存的访问请求被赋予更高的优先级, 所以设备控制器的内存访问就有可能被延迟。而在把数据从磁盘和其他辅助存储器设备传送到主存的时候, 设备控制器高速缓存可通过作为缓冲存储器来防止由于有关延迟而导致的数据丢失。设备控制器高速缓存也存在于例如软盘和光盘等某些类型的第三级存储器设备的输入/输出控制器中, 具体我们将会在下一节加以描述。另外, 在输入/输出控制器和主存之间的这种类型的数据传输可以使用我们在第 14 章中所讨论的内存直接存取 (Direct Memory Access, DMA) 技术。

A.3.4 第三级存储器和离线存储器: 可移动磁盘和磁带

在许多计算机系统中, 还存在另一附加的存储器层级, 譬如用于备份的各种类型的**磁带存储器**, 有时被称为**第三级存储器** (tertiary storage) 或**离线存储器** (offline storage, 或称为**脱机存储器**)。另外, 各种类型的**旋转式存储器**, 比如软盘、只读光盘 (CD-ROM)、读写式

① 历史上, 主存储器不一定是易失性的。特别地, 磁芯主内存即使在电源关闭时也会保持其内容。

光盘 (CD-RW)、数字化视频光盘 (DVD) 等^①，常常也被用来作为保持信息的存储介质。存储在可移动介质 (主要包括可移动磁盘和磁带) 上的信息往往由于太过庞大而不能容纳在辅助存储器上，或者通常并不经常需要或立即需要，故而并不是长期保存在硬盘上。为此，这类数据一般在计算机系统内并不经常可用，即不像在高速缓存存储器、主存和硬盘——它们被称为**在线存储器** (online storage，或称为**联机存储器**)——中的信息，后者只要在计算机系统开启的时候就可使用。

可移动介质部件可以自动化，从而使有关驱动器能够从插入驱动器的许多单独介质中进行选择，相关例子包括自动磁带库或光盘自动点唱机。在这种情况下，它们适于被称为**第三级存储器**。未被自动化的可移动介质存储部件通常被称为**脱机存储器**，因为必须经由手动方式加载有关存储介质 (软盘、数字化视频光盘、只读光盘、磁带) 后才能访问相应介质上的数据。第三级存储器和离线存储器设备也可以被视为输入/输出设备 (参见第 A.4 节)。

499

A.3.5 存储体系的管理

存储体系的各个层级之间的传输通常以多个字节或字节块 (block) 为单位来进行。主存和高速缓存存储器之间的块大小 (block size) 通常在 16 字节 (4 字) 到 256 字节 (64 字) 之间的范围内，而硬盘和主存之间的块大小通常在 4KB~16KB 之间的范围内甚至会更大些。以块为单位而不是以单字节或单字为单位进行传输的主要原因是为了通过减少总的传输时间，从而改善性能。特别是对于磁带来说，磁带运动的开启和停止操作往往会带来很大的开销。因此，每次读或写时，传输较大的块相比于传输较小的块的效率要高出许多。同样，用来访问所需信息的磁带或磁盘定位操作往往十分缓慢。一次传输更多的数据则意味着所需要的此类定位操作将会更少。

还可以通过利用局部性原理 (locality principle) 来改善性能。根据局部性原理，在任何较短的时间间隔中，程序往往倾向于访问它们的指令和操作数的一小部分。这一局部性特性已经体现在了大多数的程序当中，并且具体表现为两个方面：

- **时间局部性** (temporal locality)：根据这一特性，一道程序在访问某些存储器地址后可能很快又会再次访问相应的存储器地址。例如，循环体内的那些指令可能在短时间内被多次重复访问。
- **空间局部性** (spatial locality)：根据这一特性，如果一道程序访问了某些存储器地址，那么其可能很快会访问存储在附近的其他字。例如，指令通常被顺序地存储和访问。再比如，程序可能处理连续存储的操作数 (数据)，具体如访问连续的数组元素或顺序扫描正在编辑的文本块。

如果存储在块中的空间上邻近的多个字被加载到了高速缓存存储器中，那么在需要的情况下对后续的字访问将会是相当快的，因为它们已经存在于高速缓存中了。这种情况被称为**高速缓存命中** (cache hit)。另一方面，如果这些后续的字从未被访问，那么将它们加载到高速缓存中的成本将会被浪费。而当引用不在高速缓存存储器中的指令或操作数时，有关系统将试图在主存储器中找到它们并把它们传送到高速缓存中。这种情况被称为**高速缓存未命中** (cache miss)。

如果引起高速缓存未命中的字还不在于主存中，那么其一定存储在硬盘上，故而需要首先

① CD-ROM 代表 Compact Disc-Read Only Memory，即只读光盘存储器；CD-RW 代表 Compact Disc-Read Write，即读写式光盘；DVD 代表 Digital Video Disc，即数字化视频光盘。

500

将其传送到主存中，然后再将所需部分传送到高速缓存中。因此，有必要找到降低每存储单位的访问成本的适当的块大小。通常情况下，采用一次性传输方式在一个级别和下一个级别之间传输 n 个连续字节或字的成本要远低于使用多次传输方式来传输它们的成本。这对于在硬盘和主存之间的传输尤其如此，并且对于在主存和高速缓存存储器之间的传输而言在较小程度上也是如此。我们应当看到，操作系统的内存管理模块的主要部分就是试图优化各种类型的传输。一般来讲，操作系统负责处理硬盘和主存之间的传输，而处理器硬件负责处理内存到高速缓存的传输。

A.3.6 存储保护

存储保护 (memory protection, 或称为内存保护) 部件是主存与操作系统特别相关的另一方面。当一道正在执行的程序引用某存储器单元时，操作系统需要确保该存储器单元是该程序的地址空间 (address space) 的一部分。也就是说，应当禁止应用程序引用正在由其他程序或操作系统本身所使用的存储器单元。这样才能保护操作系统和其他用户程序及数据免受错误程序或恶意程序的破坏。

一种用于存储保护的技术是使用一对寄存器，即基址寄存器和界限寄存器 (limit register)，如图 A-5 所示。在一道程序开始执行之前，操作系统应当设置这两个寄存器以界定包含对应程序地址空间的存储器地址。用来设置基址寄存器和界限寄存器的内容的指令是特权指令，只有在操作系统内核中且处理器处于监管模式时才能使用。一旦操作系统将执行模式设置为用户模式并将控制权转移给了用户程序，那么基址寄存器和界限寄存器都将不能改变。而对相应范围之外的存储器单元的任何引用都将导致一个用来指示发生了无效存储器引用的硬件中断。每当将执行控制权转移到另一道程序时，操作系统将重新设置基址寄存器和界限寄存器。

在许多现代系统中，往往使用了更为复杂的方案。内存被划分为相等大小的内存页面 (memory page)。典型的内存页面大小一般从 512 字节到 4096 字节 (4KB) 不等。这项技术采用了页表 (page table)，而页表是用来指向可由当前正在执行的用户程序所访问的特定内存页面的数据结构。只有通过有关页表所引用的那些存储器单元才能被相应程序所访问。页表通过处理器自身内部的硬件支持来加以实现。同样，加载页表的内容的命令也是只能由处于监管模式下的操作系统内核才能执行的特权指令。我们在第 10 章和第 11 章中曾经详细讨论了

501

这种存储保护技术以及其他的存储保护技术。

A.4 输入和输出

输入和输出系统是把主存和处理器连接到其他设备的部件，它们有时被称为输入 / 输出设备 (I/O device) 或外围设备 (peripheral device)。

A.4.1 输入 / 输出设备的类型

输入 / 输出设备可被划分为四大类：用户接口设备 (user interface device)、存储设备 (storage device)、网络设备 (network device) 以及计算机控制型设备。

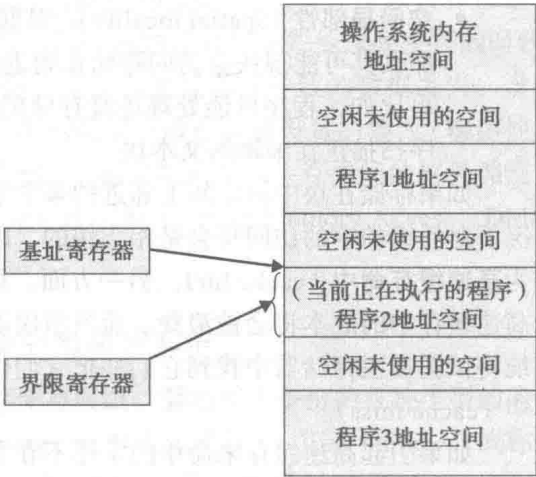


图 A-5 利用基址寄存器和界限寄存器的内存保护机制

- **用户接口型输入 / 输出设备 (user interface I/O device)**: 它们用来实现用户与计算机系统之间的交互。服务于用户和系统之间直接交互的设备包括键盘、指针式设备 (例如鼠标、轨迹球、触摸屏或触摸板)、操纵杆、话筒 (语音或声音输入)、用于输入的其他类似部件, 以及视频监视器、扬声器 (语音或声音输出) 和用于输出的其他类似部件。还有其他的一些输入 / 输出设备支持间接交互, 例如用于视频或图像输入的数码相机和扫描仪以及用于硬拷贝或胶片输出的打印机和绘图仪。
- **存储型输入 / 输出设备 (storage I/O device)**: 它们用于存储信息, 故而被认为既是输入 / 输出设备, 又是存储体系的组成部分。它们具体包括磁盘 (硬盘或软盘)、光盘 (optical disc) / 数字化视频光盘、磁带、闪存芯片, 等等。
- **网络型输入 / 输出设备 (network I/O device)**: 它们用于将计算机系统连接到网络上, 具体包括模拟电话调制解调器、数字用户线路连接、电缆调制解调器和有线电视。此外, 诸如红外线或蓝牙之类的无线连接正变得越来越普遍, 它们可以使用安装在计算机或设备中的无线网卡来连接到无线集线器上, 而无线集线器又进一步连接到网络上, 或者它们也可以直接将设备连接到计算机上。
- **受控型设备 (controlled device)**: 计算机经常被用来控制非计算型设备, 具体包括电机、暖气、空调、照明显示器, 等等。嵌入式计算机系统也可归为这一类型。

正如我们可以看到的那样, 存在各种各样的输入 / 输出设备, 并且不时地会引入新的设备。为了应对输入 / 输出设备的这种激增问题, 人们试图努力实现单一接口的标准化, 以便能够接入各种不同类型的输入 / 输出设备。通用串行总线 (Universal Serial Bus, USB) 2.0 标准就是这样的一种标准, 其支持 4.80 亿比特每秒 (bits per second, bps) 的输入 / 输出传输速率, 故而适用于连接从键盘到数字摄像机或外部硬盘驱动器等各种设备。另一种标准是 IEEE 1394, 其也允许高达 4.00 亿比特每秒的传输速率并且用于连接相同类型的设备。这一接口有两个被人熟知的专利名称, 分别是苹果公司的 FireWire (火线) 和索尼公司的 i.Link (爱点连接)。对于较高速度的设备而言, 火线接口比通用串行总线接口更为高效, 故而火线接口通常用于连接摄像机, 并被选为用于音频 / 视频部件通信和控制的标准连接接口。

A.4.2 设备控制器和设备驱动程序

设备控制器 (device controller) 是将输入 / 输出设备连接到计算机处理器和内存的接口部件。设备控制器往往包含了自己的具有专用指令集的处理器, 而设备制造商常常使用该指令集来编写用于控制输入 / 输出设备的程序。设备控制器通常还拥有**命令集 (command set)**, 即操作系统可以经由系统总线发送给控制器用以控制输入 / 输出设备的命令的集合。这些命令往往仅限于由操作系统的**设备驱动程序 (device driver)** 使用, 并且通常不能由应用程序或系统程序员所访问。此外, 许多设备控制器还具有称为**控制器高速缓存 (controller cache, 或称为控制器缓存)**的存储器部件 (参见第 A.3.3 节)。

一方面, 诸如通用串行总线接口和火线接口之类的标准设备控制器通常可用于连接到支持相应标准的任何类型的输入 / 输出设备上。另一方面, 一些专用的设备控制器 (例如磁盘控制器或图形视频控制器) 则只能连接到对应设计所支持的单一类型的输入 / 输出设备上[⊖]。

⊖ 在某些情况下, 控制器仅仅限于连接某种类型的设备的子集, 例如, ATA 控制器仅可与 ATA 磁盘驱动器而不是所有类型的磁盘驱动器协同工作。有时控制器只能使用来自单一制造商的设备, 或者甚至仅可使用特定型号的设备。

一般来说, 控制器用来处理与输入 / 输出设备之间的交互, 并且在数据从计算机主存传输到设备或者从设备传输到主存的时候, 有关控制器可以使用自己的存储器来缓冲 (或缓存) 有关数据。同时, 控制器的命令集往往包含有用来启动输入 (或输出) 操作的命令。例如, 硬盘控制器通常拥有命令用来实现针对特定磁盘块地址的盘块读取操作的启动, 有关命令同时并需提供用于保存对应盘块的计算机内存缓冲地址。图 A-6 是用来说明相关概念的简化示意图。

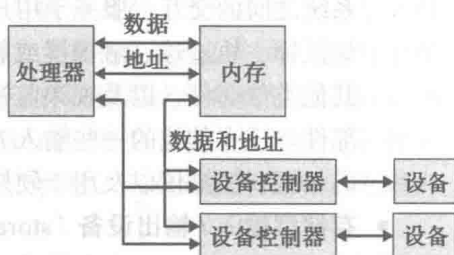


图 A-6 输入 / 输出设备经由设备控制器连接到内存和处理器的方式

在计算机这一边, 操作系统则往往会处理与设备控制器之间的所有交互。正如刚才所提到的, 操作系统中用来与设备控制器进行交互以及处理输入 / 输出的模块称为**设备驱动程序**。各设备驱动程序应当被设计成用于处理特定设备控制器的低级硬件指令和详细信息。换句话说, 设备驱动程序将会为操作系统的其他部分呈现有关设备的一个抽象和统一的视图。

503

A.4.3 输入 / 输出设备的其他分类及连接

还可以采用其他方法来对输入 / 输出设备进行分类。一种分类方法是根据设备与计算机之间的连接类型而将它们划分开来。进一步说, 输入 / 输出设备在硬件连接级别方面往往采用串行或并行的物理连接 (通常为电缆) 方式连接到内存和处理器上。其中, **串行 (serial)** 连接通过单一的一根线以串行方式来传输二进制位, 而**并行 (parallel)** 连接则通常利用多根线以每次 8 位 (或甚至更多的) 二进制位的方式来并行地传输。与简单输入 / 输出设备 (譬如键盘、鼠标或调制解调器) 的接口通常采用串行连接方式, 而诸如硬盘小型计算机系统接口之类的高速设备的连接则往往采用并行电缆方式。需要说明的是, 尽管通用串行总线和火线的控制器均采用串行电缆, 但由于其电缆是高等级和有屏蔽防护的, 故而可以支持相关控制器的高速的数据传输。

另一种更高级别的分类方法是把输入 / 输出设备划分成每次传输多个字节的**块设备 (block device)** 和每次传输单个字符或字节的**字符设备 (character device)**。磁盘是一种典型的块设备, 而键盘则是一种典型的字符设备。

第三种分类方法是根据相关连接是通过有线方式还是无线方式来进行划分。当前, 便携式计算机与网络之间的连接或者是与诸如打印机之类的输出设备之间的连接, 就越来越多地采用了无线连接方式。

A.5 网络

许多计算机往往都会连接到某种网络上。在抽象层面上, 可以认为网络连接就类似于计算机的处理器和内存可以连接到输入 / 输出设备的方式。然而, 网络允许计算机连接到其他的计算机上, 也允许计算机与连接到网络上的其他设备进行连接。故而, 这种连接支持用户访问其他计算机上的功能和信息, 并使用他们自己的计算机上所没有的设备, 同时这种连接还允许在不同计算机上运行的进程之间交换信息。

A.5.1 客户端 - 服务器模型、对等模型及多层模型

一种常见的方式是通过**客户端 - 服务器模型 (client-server model)** 来审视网络交互。其

间，一台计算机——通常是用户所在的计算机——被称为客户端 (client)。客户端可以访问一台或多台服务器 (server) 计算机以访问有关服务器所提供的信息或其他功能。服务器可能包括以下任何功能：

- 包含有大量信息的数据库服务器。
- 支持客户端访问互联网上的文档的万维网服务器 (Web server, 或称为网站服务器)。
- 支持用户在各种打印机上进行打印的打印机服务器。
- 用来管理用户文件的文件服务器。
- 用于存储和转发电子邮件的电子邮件服务器。
- 支持诸如文字处理或电子表格之类的应用程序的服务器。

关于网络交互的另一种模型是对等模型 (peer-to-peer model)，其中所有的计算机都被认为是地位平等的。例如，相关计算机可以协作解决一个大型的计算问题，且对应问题应被设计成是通过网络上的多台计算机以分布方式运行而加以处理。

随着分布式系统的发展，常常有必要构建比上述模型更为复杂的模型。大型的应用程序通常被设计成多层 (multiple tier) 模型。在典型的三层设计方案中，往往会有一个负责用户界面的前端，一个包含有关应用主要逻辑——通常称为业务规则 (business rule)——的中间层 (middle tier) 以及一个负责有关应用所有数据存储的数据库层 (database tier)。在第 17 章中，我们曾经讨论了演化形成这些更为复杂的架构的基本理由。而相关模型则在关于网络的第 15 章以及关于分布式处理系统的第 7 章和第 17 章曾有过较为详尽的讨论。

504

A.5.2 网络控制器、路由器和域名服务器

类似于计算机与用来控制输入 / 输出设备的设备控制器之间的交互方式，处理器和内存通过网络接口控制器 (Network Interface Controller, NIC) 连接到网络上。在硬件层级 (hardware level)，存在着各种类型和不同速度的网络连接，并且一直在不断地引入各种用于连接的新型技术。将计算机连接到网络的一些常见的硬件设备和技术包括调制解调器、以太网、数字信号线路、电缆调制解调器以及若干无线技术。

在物理层级，区分用于构建网络的两种类型的连接——有线方式和无线方式——是非常有用的。用于有线网络 (wired network) 的硬件包括不同类型的电缆或光纤、网络网关、路由器、交换机、集线器以及其他类似的部件。而无线网络 (wireless network) 部件则包括卫星、用于无线连接的基站、无线集线器以及红外端口和蓝牙端口等。

网络可以通过利用网桥 (bridge)、交换设备 (switching device) 或路由器 (router) 将消息从其源端点按照规定线路传送到其目的端点。为了解决管理方面的复杂性，通常将一个组织机构内部的网络划分为若干子网络，且每个子网络通过局域网 (Local Area Network, LAN) 来连接少量的计算机。这些子网络通过本地路由器 (local router) 相互连接，并通过本地路由器连接到区域路由器 (regional router) 上，而对应的区域路由器则经由一个或多个附加的互联网路由器 (Internet router) 与全球网络的其余部分进行连接。

对于互联网来说，网络上的每台计算机均拥有一个数字式网际协议 (Internet protocol, IP) 地址 (例如 192.168.2.1)，该地址用来唯一地标识相应的计算机，并且支持有关网络把寻址到该网际协议地址的消息按规定线路传送给对应的计算机。与此同时，计算机也拥有一个唯一的名称，譬如 ourserver.example.com。而称为域名服务器 (Domain Name Server, DNS) 的专用服务器拥有相关数据库，可以支持在给定计算机名称时找到其对应的数字式网

际协议地址。在此基础上, 连接在网络上的其他的专用计算机, 即路由器和交换设备, 就可以根据目的端点的数字式网际协议地址或介质访问控制 (Media Access Control, MAC) 地址找到相应的路径, 进而通过网络而把消息传送到目标计算机。期间, 这些设备使用各种级别的特定的网络协议来物理地传递消息。为说明有关概念, 图 A-7 给出了一个简化的示意图。

505

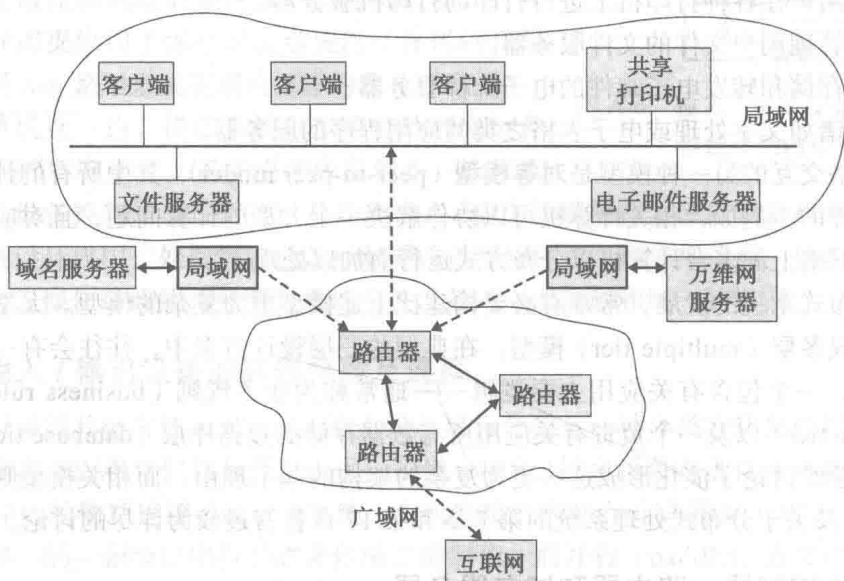


图 A-7 网络与各种计算机的连接

A.5.3 网络分类

我们以各种网络类型的传统特征描述来对这里关于网络的概要介绍加以总结^①。

局域网 (Local Area Network, LAN) 是指通常对位于有限地理区域范围内——例如, 某组织机构的一组办公室内或一栋建筑物内或多栋相邻的建筑物内——的计算机进行连接而形成的网络。有关网络主要基于穿过建筑物和在建筑物之间延伸的电缆而构建形成, 也可能包括连接到每层 (或每组相邻办公室) 的各种网络的交换机或路由器。需要说明的是, 越来越多地使用无线接入点 (wireless access point) 来支持计算机通过无线网卡而连接到局域网上。

另一方面, **广域网** (Wide Area Network, WAN) 通常是指对分布在较广阔的地理区域内的计算机进行连接而形成的网络。有关网络采用电话线、光纤电缆、卫星和其他连接部件而将成千上万个局域网彼此连接到一起, 从而支持全球范围内的计算机的相互连接。

移动网络 (mobile network) 由成千上万个作为固定基站进行运营的电信塔和控制系统所组成, 而这些基站又连接到了局域网或广域网上。诸如手机或手持式计算机或个人数字助理之类的移动设备可以连接到一个邻近的基站上, 并通过该基站与网络的其余部分相连接, 同时与全球网络的其他部分相互连接到了在一起。

506

A.6 系统结构详图

最后, 我们以图 A-8 结束本附录, 该图就我们在附录中所讨论的各种系统部件之间的相

^① 局域网和广域网之间的技术区别会有些不同, 详见第 15 章相关内容。

互连接给出了更为详细的展示。

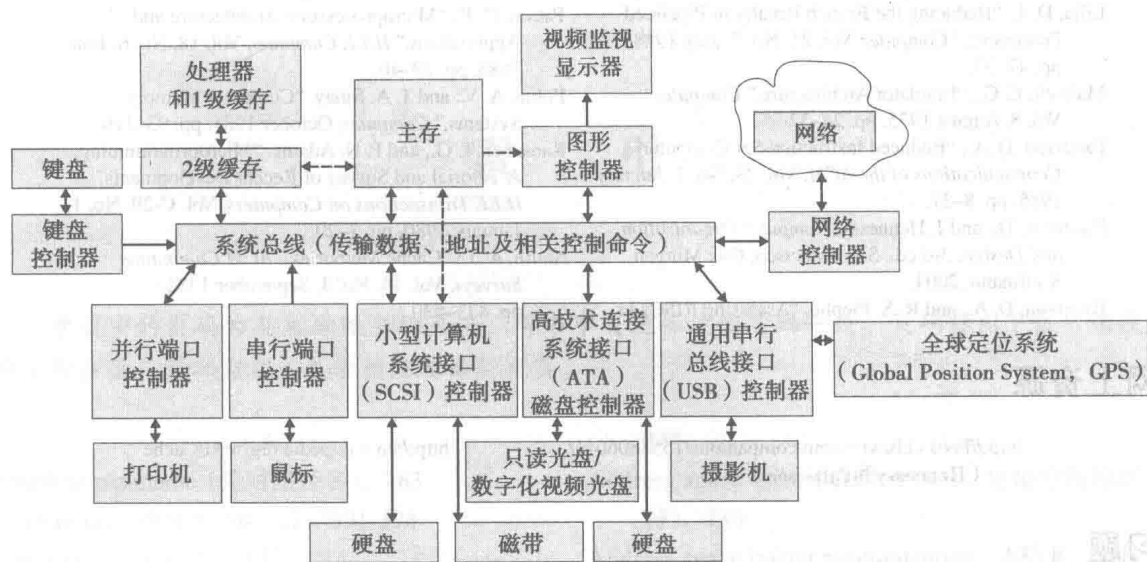


图 A-8 计算机系统组成结构详图

A.7 小结

在本附录中，我们概述了计算机系统的基本组成部件。我们从典型计算机系统组成部件的一个简单概述和结构图出发，最终总结形成了一个较为详细的——尽管仍然简化的——结构图。在其间，我们就现代计算机系统的主要组成部件包括处理器、内存和存储体系、输入/输出设备以及网络，按照一节介绍一个部件的方式分别进行了解释说明。从论述过程中，相关内容彼此之间无疑会存在某些重复。例如，硬盘既可以被认为是输入/输出设备，也可以被认为是存储体系的组成部分，同时，计算机的网络接口也可以被抽象和认为是输入/输出设备。尽管如此，传统的划分方案对于很好地结构化及编排组织我们关于计算机系统和操作系统的相关讨论和陈述却是非常有用的。

507

参考文献

- Belady, L. A., R. P. Parmelee, and C. A. Scalzi, "The IBM History of Memory Management Technology," *IBM Journal of Research and Development*, Vol. 25, No. 5, September 1981, pp. 491-503.
- Brown, G. E., et al., "Operating System Enhancement through Firmware," *SIGMICRO Newsletter*, Vol. 8, September 1977, pp. 119-133.
- Bucci, G., G. Neri, and F. Baldassarri, "MP80: A Microprogrammed CPU with a Microcoded Operating System Kernel," *Computer*, October 1981, pp. 81-90.
- Chow, F., S. Correll, M. Himmelstein, E. Killian, and L. Weber, "How Many Addressing Modes Are Enough?" *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, October 5-8, 1987, pp. 117-122.
- Davidson, S., and B. D. Shriver, "An Overview of Firmware Engineering," *Computer*, May 1978, pp. 21-31.
- DeRosa, J., R. Glackemeyer, and T. Knight, "Design and Implementation of the VAX 8600 Pipeline," *Computer*, Vol. 18, No. 5, May 1985, pp. 38-50.
- Elmer-DeWitt, P., and L. Mondi, "Hardware, Software, Vaporware," *Time*, February 3, 1986, p. 51.
- Fenner, J. N., J. A. Schmidt, H. A. Halabi, and D. P. Agrawal, "MASCO: The Design of a Microprogrammed Processor," *Computer*, Vol. 18, No. 3, March 1985, pp. 41-53.
- Foley, J. D., "Interfaces for Advanced Computing," *Scientific American*, Vol. 257, No. 4, October 1987, pp. 126-135.
- Foster, C. C., and T. Iherall, *Computer Architecture*, 3rd ed., New York: Van Nostrand Reinhold, 1985.
- Fox, E. R., K. J. Kiefer, R. F. Vangen, and S. P. Whalen, "Reduced Instruction Set Architecture for a GaAs Microprocessor System," *Computer*, Vol. 19, Issue 10, October 1986, pp. 71-81.
- Hunt, J. G., "Interrupts," *Software—Practice and Experience*, Vol. 10, No. 7, July 1980, pp. 523-530.

- Leonard, T. E., ed., *VAX Architecture Reference Manual*. Bedford, MA: Digital Press, 1987.
- Lilja, D. J., "Reducing the Branch Penalty in Pipelined Processors," *Computer*, Vol. 21, No. 7, July 1988, pp. 47–53.
- Mallach, E. G., "Emulator Architecture," *Computer*, Vol. 8, August 1975, pp. 24–32.
- Patterson, D. A., "Reduced Instruction Set Computers," *Communications of the ACM*, Vol. 28, No. 1, January 1985, pp. 8–21.
- Patterson, D., and J. Hennessy, *Computer Organization and Design*, 3rd ed., San Francisco, CA: Morgan Kaufmann, 2004.
- Patterson, D. A., and R. S. Piepho, "Assessing RISCs in High-Level Language Support," *IEEE Micro*, Vol. 2, No. 4, November 1982, pp. 9–19.
- Patton, C. P., "Microprocessors: Architecture and Applications," *IEEE Computer*, Vol. 18, No. 6, June 1985, pp. 29–40.
- Pohm, A. V., and T. A. Smay, "Computer Memory Systems," *Computer*, October 1981, pp. 93–110.
- Rauscher, T. G., and P. N. Adams, "Microprogramming: A Tutorial and Survey of Recent Developments," *IEEE Transactions on Computers*, Vol. C-29, No. 1, January 1980, pp. 2–20.
- Smith, A. J., "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, September 1982, pp. 473–530.

网上资源

<http://books.elsevier.com/companions/1558606041/>
(Hennessy和Patterson)

<http://en.wikipedia.org/wiki/Cache>

习题

- A.1 关于处理器设计的两种主要类别是什么？
- A.2 就操作系统设计和开发的讨论而言，指令集体系结构有何重要意义？
- A.3 为什么系统硬件定时器对操作系统非常重要？
- A.4 中断的目的是什么？
- 508 A.5 多核处理器芯片的意义是什么？
- A.6 计算机中的主存储器总是由电子存储器电路组成的。这是否正确？
- A.7 缓存机制对操作系统性能极为重要，无论怎样强调也不为过，也很难夸大。
- a. 缓存的目的是什么？
- b. 缓存功能所依赖的理论基础是什么？
- A.8 从理论上讲，我们可以让在辅助存储器和主存储器之间的缓存与辅助存储器一样大。这将会大大减少有关的延时。但为什么我们不这样做呢？
- A.9 内存保护的目的是什么？
- A.10 拥有设备控制器的目的是什么？
- A.11 为了帮助我们讨论和理解诸如输入/输出设备之类的复杂主题，我们可以把有关主题作为一个具有多个维度的空间来看待。我们首先根据设备的目的讨论了关于输入/输出设备的一种宽泛的分类。期间所讨论的三种宽泛的目的是什么呢？请分别给出每种类别的一些例子。
- A.12 我们还根据相关接口是通用接口还是特定设备类型接口而把输入/输出设备空间进行了划分。请分别给出每种类别的一些例子。
- A.13 与不使用内存直接存取型控制器的控制器相比，内存直接存取型控制器可以使得由于在设备和主机之间数据块的传输所引发的中断次数大大减少。除了可以明显地把处理器解放出来做其他事情之外，我们需要使用内存直接存取型控制器的理由还有哪些呢？
- A.14 设备驱动程序的功能是什么？我们如何配置操作系统来使用正确的驱动程序呢？
- A.15 利用什么机制来将诸如 `omega.example.com` 之类的计算机名称转换为网际协议地址以便在网络中使用呢？
- 509

索引

索引中的页码为英文原版书的页码,与书中页边标注的页码一致。页码标以 f 表示相应图示所在页;页码标以 t 表示相应表格所在页。

A

Absolute pathname (绝对路径名), 263
Abstraction (抽象), 50, 82, 391-394
Abstraction layers (抽象层), 14
Abstract virtual machines (抽象虚拟机), 393-394
Access bit (访问位, 或称为存取位), 242
Access Co. Ltd. (Access 有限公司), 480
Access control list (ACL, 访问控制表), 250, 369, 369f
Access control matrix (ACM, 访问控制矩阵), 368-369, 368f
Access methods (存取方法, 或称为访问方法), 265-269, 266f
Accumulator (累加器), 492
Accumulator extension (累加器扩展), 492
Acknowledgment (ACK, 确认), 404, 462
Active partition (活动分区), 307
Active runqueue (有效运行队列, 或称为活动运行队列), 448, 448f
Acyclic graph directories (无环图目录结构), 261-262
Add operation (加法运算), 488f
Address (地址)
 of data value (数据值的地址), 487
 of instruction (指令的地址, 或称为指令地址), 487
 registers (地址寄存器), 492
 space (地址空间), 425-426, 425f, 501
Addressing modes (寻址方式), 487, 488-489f, 490
Address management unit (AMU, 地址管理单元), 103
Address resolution protocol (ARP, 地址解析协议),

349

Address space identifiers (ASIDs, 地址空间标识符), 230
Address space layout randomization (ASLR, 地址空间布局随机化), 419
Advanced communication models (高级通信模型), 478
Advanced configuration power interface (ACPI, 高级配置电源接口), 441
Advanced encryption standard (AES, 高级加密标准), 375
Advanced memory management (高级的内存管理), 225-252
 purpose of (高级内存管理的目的), 225-226
 types of (高级内存管理的类型), 248-252
A5world (A5 定位区), 94
Aged shortest seek time first (ASSTF, 请求时变式最短寻道时间优先调度算法), 317
Aging (老龄化技术), 157, 242
Alarm clock (闹钟), 92
Alert box form (警报框窗体), 81f, 83
Algorithms (算法), 160-163, 322
Alias (别名), 261, 435
Allocation blocks (分配块, 即簇), 58
Allocation map (分配映射), 57
Allocation of memory (内存分配), 76-78, 77-79f, 132
Alpha (阿尔法处理器), 446
ALTO (阿尔托计算机系统), 90
Amateur Radio (AX25, 业余无线电联盟数据链路层协议), 457
Amdahl's law (阿姆达尔定律), 131

- AMD x64 (爱慕德公司 64 位处理器), 418
- AMX Multitasking Executive (AMX 多任务执行系统), 69n
- AMX OS (AMX 操作系统), 473
- Andrew File System (AFS, 安德鲁文件系统), 399
- Anomaly detection (异常检测), 379
- ANSI (美国国家标准化组织), 335
- Anticipatory scheduler (预测式调度程序), 320-321, 458
- Apache Web (阿帕奇网络), 110
- AppArmor (Application Armor, 应用程序盔甲式访问控制系统), 462
- APPC (程序与程序间高级通信), 456
- Apple, Inc (苹果公司)
- Lisa (苹果公司 Lisa 计算机系统), 90, 92
 - Mac (苹果公司 Mac 操作系统), 104
 - Script (苹果脚本语言), 102
 - Share (苹果共享机制), 98, 102
 - 68030 (苹果 68030 处理器), 102
 - 68040 (苹果 68040 处理器), 102
 - Talk (苹果对话协议), 98, 102, 108, 440, 456
- Applets (小应用程序), 365-366, 365f
- Application (应用程序), 59-60
- Application Armor (AppArmor, 应用程序盔甲式访问控制系统), 462
- Application heap (应用程序堆), 95-96
- Application layer (应用程序层), 334, 338-341, 339f
- Application logic (应用逻辑), 388
- Application partition (应用程序分区), 94, 95f
- Application processes (应用程序进程), 74
- Application program (应用程序), 59-60
- Application program interface (API, 应用程序接口), 7, 70, 395, 417
- Application programmers (应用程序员), 6-7, 7f
- Application programmers' view (应用程序员视图), 9
- Application specific integrated circuits (ASIC, 专用型集成电路), 347
- Application users (应用型用户), 6
- Application virtual machines (应用级虚拟机), 39-40
- Architectural approach to building OS (操作系统构建方法), 33-35, 34f, 35f
- Architecture of Palm (Palm 操作系统体系结构), 72f
- Archive bit (归档位), 372
- ARCNET™ (附属资源计算机网络协议), 337, 346, 348
- Arithmetic and logic unit (ALU, 算术逻辑部件), 490
- ARM processor (ARM 处理器), 476, 480, 486
- ASCII (美国信息交换标准代码), 339, 340, 364
- Assembly language (汇编语言), 491
- Associated reference count (所关联的引用计数), 243f
- Associative memory (关联存储器, 又称为联想存储器), 228
- Asymmetric algorithm (非对称算法), 375
- Asymmetric key encryption (非对称密钥加密), 375-376, 376f
- Asymmetric multiprocessing (非对称多处理), 105, 125, 194
- Asynchronicity (异步性), 5
- Asynchronous attributes (异步通信属性), 185, 187-188
- Asynchronous event (异步事件), 494
- Asynchronous read (异步读取方式), 188
- Asynchronous transfer mode (ATM, 异步传输模式), 348, 441, 457
- ATA (高技术连接系统接口, 或称为高技术附加装置), 439, 502n
- Athene (Athene 操作系统), 38
- Athlon (速龙处理器), 137
- At least once semantics (至少一次性语义), 408
- At most once semantics (至多一次性语义), 408
- Atomic (原子化, 或称为原子性质的), 191-192
- AT&T (美国电话电报公司), 114
- Attribute list (属性列表), 431
- Attributes (属性), 428
- Audio I/O (音频输入/输出), 472
- Authentication (认证), 367-368
- Authorization (授权), 368-370, 368f, 369f
- Automatic page limit balancing (页面限值自动平衡), 245, 246f
- Auto mounting (自动挂载), 288
- Average response time (平均响应时间), 160
- Average turnaround time (平均周转时间), 160

- Average wait time (平均等待时间), 161
- Avoidance (死锁避免), 203-204
- Axis Communication (安讯士公司), 446
- ## B
- Backbone (主干网), 348
- Background (后台), 63, 243
- Backing store (后备存储器), 243
- Backups, system (备份, 系统), 371-372
- Backward compatibility (向后兼容, 或称为向下兼容), 41
- Banker's Algorithm (银行家算法), 204
- Banyan (Banyan 系统), 345
- Base file record (基本文件记录), 286
- Base priority class (基本优先级类别), 423
- Base register (基址寄存器), 213, 492, 501
- Base register addressing (基址寄存器寻址), 487
- Basic input/output system (BIOS, 基本输入/输出系统), 13, 49-50, 82, 96
- Basic memory management (基本的内存管理), 209-222
- binding model of (基本的内存管理的绑定模型), 210-211
- purpose of (基本的内存管理的目的), 209-210
- 见 Memory management
- Basic rate interface (BRI, 基本速率接口), 351
- Batch files (批处理文件), 59
- Batch-oriented job streams (面向批处理的作业流), 164
- Be Inc. (Be 公司), 38, 110
- Bell Laboratories (贝尔实验室), 114
- BeOS (BeOS 操作系统), 38, 110
- Berkeley Sockets (伯克利套接字), 474
- Best efforts functionality (“尽力而为”功能特性), 341
- Best fit (最佳适应算法), 219
- Bi-directional attributes (双向连接属性), 184, 185
- Big Kernel Lock (大内核锁), 463
- Bin (程序二进制文件), 491
- Binary semaphores (二值型信号量), 193
- Binary trees (B-trees, 二叉树), 268
- Binding model (绑定模型), 210-211
- Biovine spongiform encephalopathy (BSE, 牛海绵状脑病, 或称为疯牛病), 144
- Bit (二进制位, 或称为比特, 或简称位), 496
- Bit-interleaved parity (位交叉奇偶校验), 310
- Bitmap (位图), 271-272, 273f, 430
- Bit string (位串), 486
- Blackberry (黑莓手机), 473
- Block (盘块), 269, 309
- Block devices (块设备), 299, 455, 504
- Blocked state (阻塞状态), 27
- Blocking (阻塞), 99
- Blocking I/O (阻塞式输入/输出), 187
- Block interleaved distributed parity (RAID 5, 块交织分布式奇偶校验), 311, 311f
- Blocks (块), 54, 500
- Block striping (块数据条纹划分技术), 310f
- Blue Gene (“蓝色基因”项目), 144
- Blue Matter (“蓝色物质”项目), 144
- Bluetooth (蓝牙协议), 74, 353, 446, 478
- Body area networks (BANs, 身体区域网络, 或简称体域网), 353
- Bootable partitions (引导分区), 306
- Boot block (引导块), 307
- Booted (被引导), 76
- Booting (引导), 56
- Bootstrap loader (引导装入程序), 16
- Border gateway protocol (BGP, 边界网关协议), 342
- Botnet (僵尸网络), 363
- Bots (僵尸机器), 363
- Bottom-half organization (下半部组织结构), 120
- Bounded-buffer problem (有界缓冲区问题), 195-196
- Bounds checking (边界检查), 364
- Branch operation (分支操作), 487
- Bridge (网桥), 346-347, 353, 505
- Broadcast (广播式传送), 184, 186
- Broadcast packets (广播式数据包), 335, 336
- Broadcast storms (广播风暴), 348
- Brute force attack (蛮力攻击, 或称为穷举式攻击), 374
- BSD Secure Levels (伯克利软件发行版安全分级系统), 462
- BSD UNIX File System (伯克利软件发行版 UNIX 文件系统), 453
- Buddy system (伙伴系统), 451
- Buffering (缓冲技术), 300, 304-305, 305t

- Buffering strategy (缓冲策略), 184, 185
 - Buffer overflows (缓冲区溢出), 364-365
 - Buffer overrun (缓冲区溢出), 364-365
 - Built-in-functions (内置功能), 59
 - Bully algorithm (强者算法), 402, 403, 403f
 - Burst errors (突发错误), 308
 - Business rules (业务规则), 388
 - Busy-waiting (忙等待), 193
 - Bytes (字节), 496
- C**
- Cable modems (有线电视网络调制解调器, 或称为有线调制解调器或电缆调制解调器), 351
 - Cache hits (高速缓存命中), 500
 - Cache memory (高速缓存存储器, 或称为高速缓存), 497-498
 - Cache misses (高速缓存未命中), 500
 - Caching (高速缓存), 301
 - Calculator (计算器), 92, 93f
 - Cameras (照相机, 或称为相机), 473
 - Capability list (CL, 权限表), 369, 369f
 - Carbon API (碳式应用程序接口), 109
 - Card Bus (插件总线), 438
 - Card deck (loadable program tape, 卡片叠/可加载程序带), 16f
 - Cards (插卡, 或称为卡), 51, 70
 - Care-of address (转交地址), 396
 - Carrier sense multiple access/collision detection (CSMA/CD, 载波侦听多路访问/冲突检测), 346
 - Carnegie Mellon University (卡内基梅隆大学), 399
 - Case studies (实例研究), file systems (文件系统), 284-288
 - Categorization (分类), 503
 - Category (Cat, 非屏蔽型双绞线布线的质量类别), 352
 - Cathode ray tube (CRT, 阴极射线管), 72
 - CDFS (只读光盘标准格式文件系统), 428
 - Centralized lock server (集中式锁定机制服务器), 401f
 - Central processing unit (CPU, 中央处理器/处理器)
 - components of (处理器部件), 490-491, 490f
 - defined (关于处理器的解释), 484
 - development of (处理器的开发), 103-104
 - types of (处理器的类型), 22-23
 - CERN (欧洲粒子物理研究所), 140
 - Certificate (证书), 377
 - Certificate authority (认证机构), 377
 - Cfq scheduler (complete fair queuing scheduler, 完全公平排队调度器, 或称为完全公平排队调度程序), 321
 - Change a message (篡改消息), 373
 - Change directory (cd 或 chdir, 改变目录), 264
 - Change journal (关于变更日志的特殊文件类型), 434
 - Change logging (更改日志, 即把更改操作记录到日志), 434
 - Change mode (更改文件模式), 121
 - Character devices (字符设备), 299, 455, 504
 - Character recognition (字符识别), 73
 - Checkdisk (磁盘检查修复工具例程), 293
 - Checksumming (校验和), 138
 - Child process (子进程), 123, 164-165
 - Chip-level multiprocessor (CMP, 芯片级多处理器), 124
 - Chmod (关于更改文件模式的例程或命令), 121-122
 - Choices (Choices 操作系统), 38
 - CHS addressing (柱面磁头扇区寻址, 或称为 CHS 编址), 303
 - Circular buffer (循环缓冲), 188
 - Circular-LOOK (C-LOOK, 循环向前看调度算法), 318-319, 319f, 320f, 457
 - Circular queue (循环队列), 188
 - Circular wait (循环等待), 199, 202-203
 - City traffic deadlock (城市交通中的死锁问题), 200f
 - ClassAds (类似于分类广告一样的描述), 137
 - Classic environment (传统环境), 110
 - Classified advertisements (分类广告), 137
 - Classless interdomain routing (CIDR, 无类别域间路由), 342, 344
 - Clean page mechanisms (页面清理机制), 247
 - Clean service shutdown (清理服务后再关机), 420
 - Client (客户端, 或称为客户机), 32
 - Client server (客户端-服务器, 或称为客户机-服务器), 388, 388f, 504
 - Client stub (客户端桩例程), 396
 - Clock (时钟), 493

- Clock algorithm (时钟算法), 242
- Clock cycle (系统时钟周期), 493
- Clock synchronization (时钟同步), 400-401
- Clock time (时钟时间), 131
- Clone call flags (clone 系统调用参数所用到的标志), 123t
- Clone system call (clone 系统调用), 176-177
- Cluster (簇), 269
- Clustered multiprocessing system (集群多处理系统), 134f
- Clusters (集群), 134-135
 - concept of (集群的概念), 139-140
 - defined (关于集群的解释), 129
 - types of (集群的类型), 142-146
- Clusters of workstations (COWS, 工作站集群), 130
- CMOS memory (互补金属氧化物半导体存储器), 307
- Coax (同轴电缆), 352
- Coaxial cable (同轴电缆), 352
- Cobalt (第 6 版 Palm 操作系统的名称), 480
- CODA (内容分发体系结构), 399
- Code (编码), 210
- Code Fragment Manager (代码片段管理器), 103
- Code integrity verification (代码完整性验证), 419
- CodeWarrior (由 Metrowerks 公司提供的基于 Palm 操作系统平台的商用的集成开发环境), 476
- Coding time (编码时), 211-212
- Collision (冲突), 346
- ColorSync (一款颜色管理实用程序, 寓意色彩同步), 102
- Columbia University (哥伦比亚大学), 211
- Command interpreter (命令解释器), 11, 60, 164-166
- Command line interface (命令行接口), 60, 92
- Command queuing (命令式排队), 321
- Command set (命令集), 503
- Commercial IP Security Option (CIPSO, Linux 安全系统名称, 寓意商业网际协议安全选项), 462
- Commercial platforms (商业平台), 465t
- Common Internet file system (CIFS, 通用互联网文件系统), 183, 386, 399, 461
- Common language runtime (CLR, 通用语言运行库), 39, 365, 393
- Common object request broker (CORBA, 通用对象请求代理体系结构), 398
- Communication (通信), 182
- Communication circuits (通信电路), 473
- Communication security (通信安全), 373-377, 374-376f
- Communication threats (通信威胁), 374f
- Compact disc (CD, 光盘), 288-289, 428, 452
- Compact disc-read only memory (CD-ROM, 只读光盘存储器), 108, 308, 499
- Compact disc-read write (CD-RW, 读写式光盘), 485, 499
- Compaction (紧凑), 77, 220, 220f
- Compaq (康柏公司), 324
- Compiler (编译器), 491
- Complete fair queuing (CFQ, 完全公平排队), 458
- Complete fair queuing scheduler (cfq scheduler, 完全公平排队调度器, 或称为完全公平排队调度程序), 321
- Complex file names (复杂文件名), 435
- Complex instruction set computer (CISC, 复杂指令集计算机), 486
- Compression (压缩), 434-435
- Computational Biology Center (计算生物学中心), 144
- Computational grid (计算网格), 142-143
- Computer networks (计算机网络), 331-356
 - basics of (计算机网络基础), 333-338
 - physical layer of (计算机网络物理层), 352-354
 - purpose of (计算机网络的目的), 332-333
- Computer system, overview and architecture concepts (计算机系统、总览和体系结构概念), 483-507
 - components of (计算机系统组成部件), 484-485, 485t
 - concepts of (计算机系统的概念), 507f
 - illustration of (计算机系统图解), 507, 507f
- Computer viruses (计算机病毒), 361
- Computing grids (计算网格), 135-136
- Concurrency (并发), 387
- Concurrency protection (并发性保护), 372-373
- Conditional branch-on-equal operation (分支跳转操作), 490f

- Condition variables (条件变量), 197
- Condor Project (威斯康星大学志愿计算项目), 136, 137
- Config.sys (机器硬件配置文件名), 438
- Connectionless attributes (无连接的通信属性), 184, 185-186
- Connection oriented attributes (面向连接的通信属性), 184, 185-186, 341
- Connection super-server(监听连接的超级服务器), 461
- Console (控制台), 49
- Console command processor (CCP, 控制台命令处理程序), 50, 70
- Content addressable memory (CAM, 内容可寻址存储器), 228
- Context switch (上下文切换), 23, 30, 100
- Context switch overhead (上下文切换的开销), 163
- Contiguous allocation (连续分配), 273-275, 274f
- Controlled devices (受控型设备), 502
- Controller (控制器), 321-322, 491-492
- Controller cache (控制器高速缓存, 或称为控制器缓存), 499
- Control Panel (控制面板), 92
- Control Program/Monitor (CP/M, 早期操作系统名称, 寓意控制程序/监控者)
- abstraction, component of (CP/M 的抽象及组成), 50
 - development of (CP/M 的开发), 49
- Control unit (控制器), 491
- Convoy effect (结队效应), 162
- Cooked interface (熟食接口), 82
- Cookies (小型文本文件), 366
- Cooperating process (协作式进程), 181, 386
- Cooperative multitasking (协作式多任务), 99, 157
- Coordinator (协调器), 402
- Copper wire specifications (铜线规格), 352
- Copy command (cp/copy, 文件拷贝命令), 264
- Copy-on-write (写时复制), 176, 248, 426-427
- CORBA (Common Object Request Broker Architecture, 通用对象请求代理体系结构), 134-135
- Cornell University (康奈尔大学), 362
- Counting semaphores (计数型信号量), 194
- Coupled multiprocessors (耦合型多处理器), 124
- CP-40 (IBM 硬件级虚拟机及仿真软件包), 38
- CPU preference (处理器偏好), 133
- CPU process scheduler (处理器进程调度器, 或称为进程调度器), 27
- CPU scheduler (处理器调度程序), 27
- CPU state information (处理器状态信息), 153
- CPU utilization (处理器利用率), 161
- Crash protection (系统崩溃保护), 371-372
- Critical region (临界区), 401
- Critical section (临界区), 138, 192
- CTL/C (组合按键: CTL 按键 +C 按键), 187
- Current position (当前位置), 265
- Current record pointer (当前记录指针), 265
- Current working directory (当前工作目录), 263
- Cursor tracking mouse motion (光标跟踪鼠标移动), 8-9, 9f
- Cyclic elevator (循环电梯调度算法), 318-319
- Cyclic redundancy check (CRC, 循环冗余校验, 或称为循环冗余校验码), 138, 308, 308t, 349
- Cylinders (柱面), 302-303
- Cylindrical-LOOK (圆筒状向前看调度算法), 318-319
- D
- Database file support (数据库式文件支持), 80-81
- Database servers (数据库服务器), 31, 484
- Database storage (数据库存储), 388
- Data cache (数据高速缓存), 498
- Data encryption standard (DES, 数据加密标准), 375
- Data fetch (数据提取), 494
- Data link layer (数据链路层), 345-350
- Data path (数据通路), 492
- Datapoint Corp (数据点公司), 457
- Data redundancy (数据冗余), 432
- Data registers (数据寄存器), 492
- Data run (数据盘区), 431
- Data storage area (数据存储区), 56, 57-58
- Data stream (数据流), 435
- Data striping (数据条纹划分), 309
- Data structures (数据结构), 81
- Data value (数据值), 487
- Deadline scheduler (基于截止时间的调度程序), 321, 458

- Deadlocks (死锁)
 defined (关于死锁的解释), 182
 process management and (进程管理与死锁), 197-206, 198-200f
- Decode (解码), 494
- Decryption (解密), 374
- Dedicated parity drive (奇偶校验专用磁盘), 310-311, 311f
- De facto standard (事实上的标准, 或称为事实标准), 50, 335
- Default gateway (默认网关), 343
- Default router (默认路由), 343
- Defragmentation (碎片整理), 275, 277, 433
- Defrag utility (磁盘碎片整理实用例程), 294
- De jure standards (法定的标准, 或称为法定标准), 335
- Delayed auto-start services (延迟式自动启动服务), 420
- Del command (文件删除命令), 264
- Demand paging (请求调页, 或称为请求分页), 238-248
- Demilitarized zone (DMZ, 隔离区), 379
- Denial of service (DoS, 拒绝服务攻击), 142, 345, 363
- Dense wavelength division multiplexing (DWDM, 密集波分复用), 353
- Desk accessories (桌面附件), 92
- Detection (检测), 201
- Detection deadlocks (死锁检测), 205
- Device controller (设备控制器), 10-11, 503, 503f
- Device driver (设备驱动程序), 438-439, 503
 defined (关于设备驱动程序的解释), 11
 OS, component of (操作系统组成部分), 12
- Devices (设备), 10-11
 characteristics of (设备的特性), 298-299, 298t
 classes of (设备的分类), 298-299, 298t, 455-458
 types of (设备的类型), 502
- /dev table (设备表), 454-455
- Dictionary attack (字典式攻击), 367
- Diffie-Hellman (迪菲-赫尔曼密钥交换协议), 375, 378
- Digital Equipment Corporation (DEC, 数字设备公司), 92
- Alpha (阿尔法处理器), 64, 418
- Net (数字设备公司分层网络体系结构协议), 184, 345, 440, 456
- PDP-11 (数字设备公司开发的一款复杂指令集计算机处理器), 486
- VAX (数字设备公司开发的一款复杂指令集计算机处理器), 232, 486
- Digital Research, Inc. (数字研究公司), 49
- Digital rights management (DRM, 数字版权管理), 362
- Digital subscriber lines (DSL, 数字用户线路), 351
- Digital video disk (DVD, 数字化视频光盘), 428, 452, 499
- “Dining Philosophers” problem (“哲学家就餐”问题), 199f
- Dir command (一条关于列出目录下文件的命令 dir), 264
- Directed acyclic graphs (DAGs, 有向无环图), 261
- Directed edges (有向边), 26
- Direct memory access (DMA, 内存直接存取技术), 322-325
 characteristics of (内存直接存取技术的特点), 323
 components of (内存直接存取技术的组成), 491
 future of (内存直接存取技术展望), 324-325
- Directory (目录), 259-264
- Directory access protocol (DAP, 目录访问协议), 395
- Directory records (目录记录), 286
- Directory service (目录服务), 10, 395
- Dirty pages (脏页, 或称为脏页面), 242-243
- Discovery services (发现服务), 395
- Discrete modeling (离散建模), 161
- Disk boot area (磁盘引导区), 55, 56
- Disk cache (磁盘高速缓存, 或称为磁盘缓存), 499
- Disk class (磁盘类), 439
- Disk controller (磁盘控制器), 55
- Disk drive (磁盘驱动器), 55
- Disk format (磁盘格式), 51, 55
- Disk head (磁头), 55
- Disk management (磁盘管理), 54-55, 54f
- Disk scheduling, I/O management and (磁盘调度, 输入/输出管理), 297-325

- characteristics of (设备特性), 298-299
 - technology of (输入/输出技术), 299-302
 - Disk scheduling algorithms (磁盘调度算法), 322
 - Disk system (磁盘系统), 54-55, 54f
 - Dispatch (调度), 155
 - Dispatcher thread (调度线程), 392
 - Displacement (d, 偏移地址), 226, 234, 487
 - Display management (显示管理), 83-84, 84t
 - Distributed Component Object Model (DCOM, 分布式组件对象模型), 134, 135
 - Distributed computing environment (DCE, 分布式计算环境), 398
 - Distributed Computing Environment/Remote Procedure Calls (DCE/RPC, 分布式计算环境/远程过程调用), 461
 - Distributed file system (DFS, 分布式文件系统), 399-400
 - Distributed OS (分布式操作系统), 385-409
 - defined (关于分布式操作系统的解释), 32
 - development of (分布式操作系统的开发), 386-388, 387f
 - models of (分布式操作系统的模型), 396-400, 397f
 - Distributed processing (分布式处理), 127-147
 - architectures of (分布式处理体系结构), 133f, 134-138, 134f
 - concepts of (分布式处理的概念), 128
 - Distributed system architecture (分布式系统体系结构), 132-134, 133f
 - Distributed transactions (分布式事务), 405-406
 - Distribution (发行版本), 116
 - Distribution transparency (分布透明性), 387
 - Distributive objects (分布式对象), 398
 - DLL Hell (动态链接库地狱), 222
 - DMA controller (内存直接存取型控制器), 298, 322-323, 491
 - Domain (域), 343
 - Domain name servers (DNS, 域名服务器), 343, 505
 - Domain name system (DNS, 域名系统), 394-395
 - Dotted decimal notation (点分十进制标记), 342
 - Double buffering (双缓冲技术), 300, 301f
 - Drive rotation speed (驱动器旋转速度), 305t
 - Dual memory access (两次访问内存), 228, 228f
 - Duplexing (磁盘双工), 310
 - Dynamically loadable modules (DLMs, 动态可加载模块), 117-119, 118f, 119t
 - Dynamic bad-cluster handling (动态坏簇处理), 435
 - Dynamic host configuration protocol (DHCP, 动态主机配置协议), 343
 - Dynamic link libraries (DLLs, 动态链接库), 221-222
 - Dynamic load balancing (动态负载均衡), 32
 - Dynamic loading (动态加载), 221
 - Dynamic memory (动态内存), 6
 - Dynamic priority (动态优先级), 423-424
 - Dynamic random access memory (DRAM, 动态随机存取存储器), 76, 497
 - Dynamic relocation (动态重定位), 213
- E
- Effective access time (EAT, 有效访问时间), 229-230
 - Effective address time (EAT, 有效寻址时间), 239-240, 240t
 - 802.11 (无线局域网物理协议), 457
 - 800 TFLOPS (每秒800万亿次浮点运算操作), 137
 - 80/20 rule (80/20法则), 239
 - EISA buses (扩展版企业标准架构总线), 438
 - Election (选举), 402-404, 403f, 404f
 - Electronic main memory (电子主内存), 70
 - Elevator algorithm (电梯调度算法), 317-318
 - E-mail (电子邮件), 399
 - Embedded computer systems (嵌入式计算机系统), 484
 - Emulator (Palm 仿真器软件包), 476
 - Enabled application (志愿式启用型应用程序), 183-184, 386
 - Encrypting file system (EFS, 加密文件系统), 433
 - Encryption (加密), 374-376, 374-376f, 433-434
 - End users (终端用户), 6, 7f
 - End user's view (终端用户视图), 9
 - Enhanced interior gateway routing protocol (EIGRP, 增强型内部网关路由协议), 342
 - Enhanced second chance algorithm (增强型二次机会算法), 244
 - Entry section (进入区), 192
 - Environment (操作系统环境), 121-122, 123f,

- 394, 422
- EPOC (EPOC 操作系统, 寓意个人便利设备的新纪元), 68, 476
- Equal allocation (公平分配, 或称为平均分配), 245
- Error correction codes (ECC, 错误校验码), 307-310
- Error detection codes (EDC, 差错检测码), 307-308, 308t
- EthernetTM (以太网), 346
- ETRAX CRIS (安讯士公司 Axis Communications 开发的一款处理器), 446
- Event-driven programs (面向事件驱动的程序), 84-85, 85f
- Event loop (事件循环), 74
- Exactly once semantics (恰好一次性的语义), 408
- Exchange libraries (交换库), 474
- Exchange Manager (交换管理器), 474
- Exec system call (exec 系统调用), 166
- Executable code (可执行代码), 61, 491
- Executable programs (可执行程序), 491
- Execution, defined (关于执行过程的解释), 20
- Execution modes (执行模式), 28-29, 490
- Exit section (退出区), 192
- Exit state (终止状态), 155
- Expired runqueue (失效运行队列, 或称为过期运行队列), 448
- Exponentially decaying (按指数方式衰减), 158
- Ext2fs (Linux 第二代扩展文件系统), 452
- Ext3 (Linux 第三代扩展文件系统), 295
- Extent (扩展盘块区), 274, 429
- Extent counter (延长计数), 57
- External attributes (外部属性), 429-430
- EXternal Data Representation (XDR, 外部数据表示), 397
- External fragmentation (外部碎片化), 76, 77f, 219, 275
- ## F
- Failure, handling (故障, 处理), 142
- Fair-share scheduling (公平分配调度), 158
- Family x86 memory map (关于 x86 体系机构的内存映射), 425f
- Fast Ethernet (快速以太网), 348
- Fast user switching (快速用户切换), 415
- Fat binaries (胖二进制代码), 103
- FAT file systems (FAT 文件系统), 285f, 285t
- Fault tolerance (容错性, 或称为容错能力), 406-409, 407f, 408f, 432
- FDDI (Fiber Distributed Data Interface, 光纤分布式数据接口), 457
- FDISK (DOS 系统磁盘分区管理实用程序), 306
- Feedback (反馈), 160
- Fetch-and-add (取-加操作), 193
- Fiber distributed data interface (FDDI, 光纤分布式数据接口), 346, 457
- Fiber optic specifications (光纤规格), 352-353
- Fiber to the Curb (FTTC, 光纤入户), 351
- Fields (字段), 486
- File abstraction (文件抽象), 265
- File allocation table (FAT, 文件分配表)
- defined (关于文件分配表的解释), 284
 - 16 (FAT16 文件系统), 284, 428
 - structure of (文件分配表结构), 272
 - 32 (FAT32 文件系统), 284, 428
 - 12 (FAT12 文件系统), 284, 290, 305, 428
- File buffering (文件缓冲), 10
- File control blocks (文件控制块), 122, 123f
- File copying (文件复制), 10f
- File directory area (文件目录区), 55, 56-57
- File encryption (文件加密), 108
- File management (文件管理), 22
- File metadata (文件元数据), 259, 264
- File mode (文件模式), 122
- File name (文件名), 56-57, 430
- File open dialog box (“文件打开”对话框), 440f
- File permissions (文件访问权限, 或称为文件操作权限), 121-122
- File protection (文件保护), 373
- File record (文件记录), 284
- File redirector (文件重定向器), 453
- File servers (文件服务器), 31, 484
- File sharing techniques (文件共享技术), 141
- File storage (文件存储), 10, 278f
- File support (文件支持), 80-81, 452-454, 472
- File system (文件系统)
- basic (文件系统基础), 257-280
 - multiple (多文件系统), 290-292, 291f, 292f
 - purpose of (文件系统的目的), 258, 258f, 283
 - types of (文件系统类型), 283-295

- File transfer protocol (FTP, 文件传输协议), 32, 114, 340, 344, 415
- File type (文件类型), 56-57
- Filters (过滤器), 437
- Finder (查找器应用程序), 92, 96, 98
- Firewall (防火墙), 345, 379, 379f
- FireWire (火线), 106, 502
- Firmware (固件), 55, 496
- First come, first served (FCFS, 先来先服务调度算法), 156-157, 315, 315f
- First fit (首次适应算法), 219
- First in, first out (FIFO, 先进先出淘汰算法), 37, 241, 315, 315f, 449
- Five-state process model (五状态进程模型), 154f
- Fixed data (固定的数据), 61
- Fixed memory (固定存储器), 6
- Fixed number of processes (固定进程数), 216-218, 217f, 218f
- Fixed load address (固定的加载地址), 59
- Flash memory (闪存), 71
- Flat process group (平坦型进程组), 407, 408f
- Floating point unit (浮点部件), 490
- Floppy disk (软盘), 54f, 371
- FlProtect (利用 Win32 库函数创建内存映射对象时关于访问控制及保护设置的参数), 250
- Folder records (文件夹记录), 286
- Folders (文件夹), 93, 259
- Folding@home (“蛋白质折叠之家”), 144
- Foreground process (前台进程), 63
- Fork(用于创建进程的 fork 访管调用或系统调用), 164-165, 427
- “Forking a child”(创建一个子进程), 165
- Fork system call(用于创建进程的 fork 系统调用), 176
- Forms (窗体), 83
- Forth (Forth 程序设计语言), 476
- Fortran compiler (Fortran 程序设计语言编译器), 173
- Frac-T (fractional T1, 次 T1 线路), 350
- Fragmented file (碎片化文件), 277
- Frame (f, 页框, 或称为物理块, 有时也称为内存页面, 甚至简单地称为页面), 226, 227
- Frame relay (帧中继), 350, 350-351, 457
- FreeBSD (由加州大学伯克利分校开发的 UNIX 版衍生形成的免费开源的操作系统), 110
- Free Software Foundation (自由软件基金会), 115
- Free space (空闲空间), 269-270
- Free space chain (空闲空间链), 270f
- Free space tracking (空闲空间监测), 79-80, 269-272, 270-273f
- F-SCAN (先来先服务 - 向前看调度算法), 320
- Fsck (类 UNIX 系统中磁盘及文件系统检查修复工具实用例程), 294, 299
- Full duplex (全双工模式), 335
- Fully connected mesh (完全互联的网状网络拓扑结构), 337f
- Fully qualified name (完全限定式名称), 343
- Functional classes of OS (面向功能的操作系统分类), 29-33
- Functionality (功能), 5-6, 40-41
- Function migration (功能迁移), 349-350
- Function name (*fn) (传给系统调用的函数名参数), 176
- Functions of OS (操作系统的功能), 20-22
- ## G
- Gadgets (小工具), 83
- Garbage collection (碎片收集), 79f
- Garnet (5.4 版起 Palm 操作系统的应用程序接口名称), 480
- General-purpose registers (通用寄存器), 492
- Genome shotgun assembly approach (基因组鸟枪组合法), 143
- Gigabit Ethernet (千兆以太网), 347
- Gigabytes (GB, 笼统计为 10 亿字节, 即千兆字节), 497
- Gigahertz (GHz, 10 亿赫兹, 即千兆赫兹), 128
- GigAssembler (公共人类基因组计划集群计算程序), 143
- GIMPS (梅森素数大搜索项目计算程序), 175
- Global positioning system (GPS, 全球定位系统), 70
- Global replacement (全局置换), 244
- Globus cluster (格洛伯斯集群), 145-146
- Globus Toolkit (格洛伯斯工具包), 135
- GNU Network Object Model Environment (GNOME, GNU 网络对象模型环境), 115, 459
- Google (谷歌), 182, 389-391, 390f
- GOSSIP (一种协议), 334

- Graffiti input (涂鸦式输入), 70
- Graphical user interface (GUI, 图形化用户界面, 或称为图形化用户接口), 91-92, 96-97, 102
elements of (图形化用户界面构成要素), 83, 84
interface of (图形化用户界面接口), 418
MultiFinder (多重查找器), 100
- Graph workflows (工作流图), 130f
- Green threads (绿色线程), 170
- Grid (网格), 129-130, 140-141
- Grid computing (网格计算), 175
- Grouped free space chain (基于分组的空闲空间链), 272f
- Grouping (分组), 271
- Groups (分组), 369-370
- Guaranteed scheduling (保证型调度), 157-158
- Guard area (警戒区), 425
- Guest OS (客户机操作系统), 38, 392
- GUI (图形化用户界面, 或称为图形化用户接口)
见 Graphical user interface (GUI)
- ## H
- Hacker release (“黑客”版本, 即开发版本), 116
- Hackers (黑客), 360
- Hamming (海明码), 308
- Handspring (腾跃公司), 69
- Hard deadlines (硬性的截止时间), 32
- Hard disk (硬盘), 302f, 498
- Hardware (硬件), 82-83
- Hardware abstraction layer (HAL, 硬件抽象层), 72-73, 421
- Hardware changes (硬件变化), 106
- Hardware components (硬件组成), 51f
- Hardware details, hiding (硬件细节, 隐藏), 82
- Hardware events (硬件事件), 495
- Hardware locking instructions (硬件锁指令), 192-193
- Hardware system dependent (硬件系统相关), 163
- Hardware virtual machines (硬件级虚拟机), 38-39
- Hash (散列函数, 也称为哈希函数), 376-377
- Hashed access (散列存取, 或称为散列访问), 268
- “Have the focus”(“持有焦点”), 97
- Header (数据库首部; 报头), 80, 334
- “Head of line blocking”(“队首阻塞”), 162
- Heads (读写磁头), 302-303
- Heap (堆), 61, 76
- H8/300 (日立 Hitachi 公司的 H8/300 系列处理器), 446
- Heterogeneity (异构性), 185, 187, 387
- Hewlett-Packard (惠普公司), 446
- HFile (创建内存映射对象时的一个参数, 即指向用来创建该对象的文件的句柄), 250
- Hierarchical File System (HFS, 多级文件系统, 或称为分层式文件系统), 98
- Hierarchical File System Plus (HFS+, 多级文件系统升级版), 105-106
- Hierarchical group (分层式进程组), 407
- Hierarchical process group (分层式进程组), 407f
- Hierarchical tree (树形网络拓扑结构), 336f
- Higher-level access methods (更高级别的存取方法), 267-268, 267f
- Higher-level services (高级服务), 31-32
- Higher-level system view (高级系统视图), 8
- Highest response ratio next (HRRN, 高响应比优先调度), 159
- High performance file system (HPFS, 高性能文件系统), 428
- Hitachi (日立公司), 446
- Hit ratio (命中率), 229
- Hold-and-wait (持有 - 等待条件), 198, 202
- Hole (空穴), 219
- Home address (主地址), 396
- Homogeneous (同构通信), 185, 187
- Honeywell (霍尼韦尔公司), 31
- Horizontal distribution (水平分布), 390, 390-391
- Host OS (主机操作系统), 38, 392
- Hot standby (热备份), 314
- Hub (集线器), 346
- Human Genome Project (人类基因组计划), 143
- Hybrid hard drives (HHD, 混合式硬盘), 325, 420-421
- Hyperlinks (超链接), 398
- Hypertext Transport Protocol (HTTP, 超文本传输协议), 114, 339-340, 378
- Hyper-Threading (超线程), 172, 446
- ## I
- IBM (美国国际商用机器公司)
Computational Biology Center (IBM 计算生物学中心), 144

- CP/M, use of (IBM CP/M 操作系统, 使用), 50
 emulation packages produced by (IBM 开发的仿真软件包), 38
 MicroChannel (IBM 微通道总线), 438
 networking (IBM 网络协议), 440
 PC (IBM 个人计算机), 91, 104
 RS6000 (IBM RS6000 处理器架构), 103
 650 (IBM 650 计算机系统), 211
 360 (IBM 360 计算机系统), 275
 Idempotent (幂等的), 407
 Identifier (ID, 用户标识符), 121
 Ident service (认证服务), 463
 Idle process (空闲进程), 157
 Idle thread (空闲线程), 424
 IEEE (电气电子工程师协会)
 Committee (电气电子工程师协会委员会), 115
 802 (IEEE-802 网络), 308
 1003 (IEEE 1003 可移植操作系统接口标准), 422
 1394 (IEEE 1394 串行接口标准), 438
 i.Link (索尼公司的爱点连接串行接口), 502
 i-list (索引结点列表, 或称为索引结点集), 286-287
 Immediate addressing (立即寻址), 487
 Improved linked list (改进的链表法), 270-271, 271f, 272f
 Inconsistent state (不一致的状态), 405
 Incremental releases (增量版本), 96
 Independent data disks with double parity (独立的双奇偶校验数据磁盘), 311-312
 Independent process (独立型进程), 181
 Index block (索引盘块), 270
 Indexed access (索引存取), 267-268, 267f
 Indexed allocation (索引分配), 278-279, 278-280f
 Indexed free space chain (基于索引机制的空闲空间链), 271f
 Indexed sequential access method (ISAM, 索引顺序存取方法), 267
 Index hole (指示孔), 54
 Indexing (索引), 435
 Index register (索引寄存器), 487, 492
 Index root (索引根), 430
 Indirect addressing (间接寻址), 487
 Inetd server (Linux 操作系统用于监听连接的超级服务器), 461
 Infiniband (无限带宽), 446
 Init phase (程序的初始化阶段), 215
 Inode (索引结点), 259, 286, 287t
 Input/output (I/O, 输入/输出)
 classes of (输入/输出设备分类), 484-485
 computer systems, overview of (计算机系统, 输入/输出概述), 502-504
 Linux, use of (Linux 系统, 输入/输出使用), 454-458
 management of (输入/输出管理), 297-325
 Palm OS, subsystems of (Palm 操作系统, 输入/输出子系统), 472-473
 single process OS, management of (单进程操作系统, 输入/输出管理), 52-54
 single-user multitasking OS, basic (单用户多任务操作系统, 基本输入/输出), 82
 Windows NT, use of (Windows NT 系统, 输入/输出使用), 436-439
 Instant messaging (IM, 即时通信), 333, 373, 479
 Instruction address (指令地址), 487
 Instruction cache (指令高速缓存), 498
 Instruction counter (指令计数器), 167f, 492
 Instruction execution cycle (指令执行周期), 493-494
 Instruction fetch (指令提取), 493
 Instruction formats (指令格式), 488f
 Instruction pointer (指令指针), 166
 Instruction register (指令寄存器), 152, 492
 Instruction set (指令集), 485, 486-490
 Integrated circuits (ICs, 集成电路), 51
 Integrated development environments (IDEs, 集成开发环境), 476
 Integrated services for digital networks (ISDN, 综合业务数字网络), 351
 Intel (英特尔)
 CPU (英特尔处理器), 417
 8080 (英特尔 8080 处理器), 50
 8088 (英特尔 8088 处理器), 50
 8080/8085 (英特尔 8080/8085 处理器), 49
 80 X 86 (英特尔 80 X 86 处理器), 187
 i860 (英特尔 i860 处理器), 418
 Itanium 64 (英特尔安腾 64 位处理器), 418
 PC (英特尔处理器个人计算机), 103

- Pentium (英特尔奔腾处理器), 233
 - 386 (英特尔 386 处理器), 115, 116
 - x86 (英特尔 x86 处理器), 418, 486
 - Interactive mail access protocol (IMAP, 交互式邮件访问协议), 340-341, 344
 - Interactiveness (交互性), 449, 464
 - Interactive processing (交互式处理), 164
 - Intercept message (截获消息), 373
 - Interface description language (IDL, 接口描述语言), 397-398
 - Interface Manager (IM, 界面管理器), 416
 - Interior gateway routing protocol (IGRP, 内部网关路由协议), 342
 - Internal fragmentation (内部碎片), 217-218, 218f, 275
 - International Standards Organization (ISO, 国际标准化组织), 334, 335, 395
 - 9660 (光盘格式国际标准 ISO-9660), 288
 - 9945-1 (可移植操作系统接口标准 ISO 9945-1), 422
 - 13346 (面向可擦写的光盘和数字化视频光盘的 ISO 13346 标准格式文件系统), 428
 - Internet (互联网), 107, 333, 399
 - Internet applications (互联网应用程序), 474
 - Internet browser (互联网浏览器), 4
 - Internet control message protocol (ICMP, 互联网控制消息协议), 354
 - Internet Engineering Task Force (IETF, 互联网工程任务组), 335, 339
 - Internet Mail Consortium (互联网邮件协会), 474
 - Internet naming authority (INA, 互联网命名机构), 394-395
 - Internet Security Association and Key Management Protocol/Oakley (ISA KMP/Oakley, 互联网安全联盟和密钥管理协议), 378
 - Internet 2 (第二代互联网), 344
 - Internet Worm (互联网蠕虫), 362, 363
 - Interprocess communication (进程间通信), 133, 184-190
 - Interrupt (中断), 8, 35, 494, 495
 - Interrupt disable (中断禁用), 495
 - Interrupt handlers (中断处理程序), 119-120
 - Interrupt handling (中断处理), 35-36, 35f, 52
 - Interrupt mask (中断屏蔽), 495
 - Interrupt registers (中断寄存器), 492, 494
 - Interrupt request level (IRQ, 中断请求级), 438
 - Interrupt vectors (中断向量), 35-36, 35f
 - Interthread communication (线程间通信), 167
 - Intrusion detection systems (IDSs, 入侵检测系统), 379
 - Intrusion prevention systems (IPSs, 入侵防御系统), 379
 - I-number (索引结点编号), 286
 - Inverted page table (反置页表), 232-233, 232f
 - I/O (输入/输出) 见 Input/output (I/O)
 - Ioctl system call (用来实现硬件输入/输出控制的 ioctl 系统调用), 299
 - IP address (网际协议地址, 或称为 IP 地址), 342
 - IPC systems (进程间通信系统), 188-189
 - IP foreign agent (网际协议外部代理), 396
 - IP home agent (网际协议主代理), 396
 - IP routing (网际协议路由, 或称为 IP 路由), 342-343
 - IP security protocol (网际协议安全协议, 或称为 IP 安全协议), 378
 - IP version 6 (第 6 版网际协议), 344, 456
 - IPX/SPX (互联网数据包交换协议/序列数据包交换协议), 341, 345, 440, 456
 - IrDA (红外开发商协会), 474
 - IrOBEX (红外对象交换协议), 474
 - ISA Plug-and-Play (企业标准架构即插即用), 446
 - ISDN (Integrated Services Digital Network, 综合业务数字网络), 457
 - ISDN BRI (综合业务数字网络基本速率接口服务), 351
 - ISS (一款端口扫描软件), 462
 - ITU-T (国际电信联盟电信标准局), 395
- J
- Java (Java 程序设计语言), 177
 - RMI (远程方法调用), 134
 - Script (JavaScript 脚本语言), 365
 - Java virtual machine (JVM, Java 虚拟机), 39, 365, 393
 - JFS (日志式文件系统), 295
 - Jini (一种用于动态创建分布式系统并完成发现服务的中间件设计), 395
 - JNode (用 Java 语言编写的操作系统), 38
 - Job (作业), 26, 151

- Job Control Language (JCL, 作业控制语言), 26
- Jobs, Steven (史蒂夫·乔布斯, 苹果电脑公司创始人之一), 90
- Journaling file system (日志式文件系统), 295, 432
- Jump operation (跳转操作), 487
- Junctions (联结点), 435
- Just-in-time (JIT, 即时编译), 40, 394
- ### K
- KADAK Products Limited (柯达产品有限公司), 69n
- Keep-alive notices (“连接保持”的通知), 403
- Kerberos (一种身份认证协议), 373, 462
- Kernel (内核), 11
- Kernel approach (内核方法), 33
- Kernel architecture (内核体系结构), 446-447, 447f
- Kernel-level thread (内核级线程), 169
- Kernel mode (内核模式), 29
- Keyboard (键盘), 52-53, 72-73
- Keyboard interface chip (键盘接口芯片), 53
- Keychain Access (密码链访问), 108
- Keys (密钥), 374
- Kildall, Gary (加里·基尔代尔, 个人计算机系统 CP/M 的设计者), 49
- Kilobytes (KB, 1024 字节, 笼统计为千字节), 498
- Kool Desktop Environment (KDE, Kool 桌面环境), 459
- Ksym (内核符号表), 119
- ### L
- Lamport timestamps (兰伯特时间戳), 400
- LAN Manager for Windows legacy system (传统 Windows 操作系统所采用的局域网管理器协议), 440
- Large Hadron Collider (LHC, 大型强子对撞机), 140
- Large network operation centers (NOCs, 大型的网络运营中心), 354
- Large page tables (大页表), 231-232, 231f
- LaserWriter (苹果公司激光打印机品牌名), 98
- Last mile (“最后一英里”), 350
- LAT (本地传输协议), 345
- Lawrence Livermore Labs (劳伦斯利弗莫尔实验室), 144
- Layered architecture (层次式体系结构, 或称为分层式体系结构), 25, 33
- Layered OS approach (分层式操作系统方法), 33-34, 34f
- Layered view of OS (操作系统的分层视图), 13f
- Layer three switches (第三层交换机), 347
- Layer two switches (第二层交换机), 347
- Lazy loading (延迟加载), 238
- L1 cache (1 级高速缓存, 或称为 L1 高速缓存), 498
- L2 cache (2 级高速缓存, 或称为 L2 高速缓存), 498
- L3 cache (3 级高速缓存, 或称为 L3 高速缓存), 498
- LDAP-to-DAP gateway (从轻量级目录访问协议到目录访问协议的网关), 395
- Learning bridge (学习型网桥), 347
- Leased line (专用线路, 又称为租用线路), 348
- Least recently used (LRU, 最近最久未使用页面淘汰算法), 241
- Legacy applications (传统应用程序), 421-422
- Legacy systems (传统系统), 440
- Lightweight directory access protocol (LDAP, 轻量级目录访问协议), 395, 462
- Light-weight process (LWP, 轻量级进程), 167, 176
- Limit register (界限寄存器), 501
- Linear addressing (线性寻址), 237f
- Linear bus (总线型网络拓扑结构), 337f
- Linear topology (线形网络拓扑结构), 336f
- Link tracking (链接跟踪), 434
- Linked allocation (链接分配), 275-278, 276f, 277
- Linked index block lists (索引盘块链表), 279
- Linked indexed file (基于索引盘块链表的文件), 280f
- Linked list (链表法), 269-270, 270f
- Linking (链接), 210
- Linking time (链接时), 212
- Link state (链路状态), 342
- Linux (Linux 操作系统), 445-466
- device classes of (Linux 系统设备分类), 298t
 - file system of (Linux 操作系统之文件系统), 286-288
 - historical overview of (Linux 操作系统发展简史), 446
 - organization of (Linux 操作系统的基本结构), 446

- 116-117, 117f
 - 2.2 (Linux 内核 2.2 版), 451
 - 2.6 (Linux 内核 2.6 版), 447, 451
 - variants of (Linux 操作系统变种), 463-464, 465t
 - Linux Assigned Names And Numbers Authority (LANANA, Linux 名称和编号分配机构), 455
 - Linux Intrusion Detection System (Linux 入侵检测系统), 462
 - Linux security module (LSM, Linux 安全模块), 462
 - Liquid crystal display (LCD, 液晶显示器), 69
 - “Little endian/big endian” problem (“小端字节序/大端字节序”问题), 187
 - Load (加载), 210
 - Loadable program tape (card deck, 可加载程序带/卡片叠), 16f
 - Loader (装入程序, 或称为加载程序), 16, 61
 - Load operation (加载操作), 489f
 - Local area network (LAN, 局域网), 335-336, 338f, 506
 - Local file system (本地文件系统), 288-289
 - Locality of reference (引用的局部性), 238
 - Locality principle (局部性原理), 500
 - Localization (本地化), 106
 - Local replacement (局部置换), 244, 427
 - Local Security Authority (LSA, 本地安全授权机构), 461-462
 - LocalTalk (本地对话协议), 98
 - Location (位置), 387
 - Location aware applications (位置感知类应用程序), 479-480
 - Location transparency (位置透明性), 32
 - Lock a page (锁定一个页面), 246
 - Locks (锁), 192, 192f
 - Lock server (锁定机制服务器), 401
 - Log-based file system (基于日志的文件系统, 即日志式文件系统), 294-295, 432
 - Logical address (逻辑地址), 214f
 - Logical block address (LBA, 逻辑块地址), 102, 214, 303-304
 - Logical clocks (逻辑时钟), 400
 - Logical disk organization (磁盘逻辑组织), 305-308
 - Logical structure (逻辑结构), 259-262, 259t, 260f, 262f
 - Log-structured file system (日志式文件系统), 295
 - Longitudinal redundancy check (LRC, 纵向冗余校验码, 或称为纵向冗余校验), 308
 - Long-term scheduler (长程调度器), 164
 - LOOK (向前看调度算法), 317-318, 318f
 - Look-and-feel (外观和感觉), 459
 - Lotus Notes (企业级通信协同工作平台), 399
 - Low-level formatting (低级格式化), 104, 304
 - Low-level network access services (低级的网络访问服务), 31
 - Low-level system view (低级系统视图), 8
 - LpName (创建内存映射对象时的一个参数, 用来指定欲映射的文件的名称), 250
 - Ls command (一条关于列出目录下文件的命令 ls), 264
- ## M
- MAC (麦金塔系统) 见 Macintosh
 - MAC address (介质访问控制地址), 346
 - Machine language (机器语言), 486-490, 488-489f
 - Mach kernel (由卡内基梅隆大学开发的用于支持操作系统研究的操作系统微内核), 110
 - Macintosh (MAC, Mac 系统)
 - Classic (Mac 标配计算机系统), 96
 - File System (MFS, Mac 文件系统), 98
 - 512K (Mac 512K 机型), 96
 - II (Mac-II 型计算机), 98, 102-103
 - origin of (Mac 计算机的起源), 90-91
 - OS Standard File System (Mac 操作系统标准文件系统), 98
 - OS X (X 版 Mac 操作系统), 109-110, 399, 418
 - ROM (Mac 系统只读存储器), 96
 - SE (Mac-II 型计算机), 98
 - Mad cow disease (疯牛病), 144
 - Magnetic disk hard-drive (磁盘硬盘驱动器), 498
 - Mailbox (邮箱), 189
 - Main-loop phase (程序的主循环阶段), 215
 - Main memory (主内存, 简称主存或内存), 23, 51, 235f, 497-498
 - Major modules (主要模块), 24-25, 25f
 - Management information base (MIB, 管理信息库), 354
 - Management tools (管理工具), 354

- Many-to-many mapping (多对多映射模型), 171
- Many-to-one mapping (多对一映射模型), 170
- Mapping addresses (地址映射), 227f, 349
- Marshalling (编组), 396
- Mars Pathfinder (火星探路者项目), 195
- Master boot block (MBB, 主引导块), 437
- Master boot record (MBR, 主引导记录), 307, 437
- Master chunk pointer (MCT, 主控内存块指针), 77
- Master file table (MFT, 主文件表), 284, 286, 428-431
- Master pointer block (主指针块), 95
- Master pointer table (MPT, 主控指针表), 77, 78f
- Matchmake (匹配撮合), 137
- Maximalist philosophy (最大化哲理), 16, 40
- MD5 (一种消息摘要散列算法), 376-377
- Mean time between failures (MTBF, 平均故障间隔时间), 309
- Media access control (MAC, 介质访问控制), 346, 505
- Megabytes (MB, 1 048 576 字节, 笼统计为兆字节, 即百万字节), 498
- Memory (内存)
 - overlay (内存覆盖), 60
 - pages (内存页面), 501
 - protection (存储保护), 501, 501f
 - space (内存空间), 25
 - unit (存储器部件), 496-501, 501f
- Memory access control (内存访问控制), 230-231, 240t
- Memory address (存储器地址, 有时也称为内存地址), 438, 496
- Memory allocation (内存分配), 76-78, 77-79f, 132
- Memory areas (内存区域), 236
- Memory bus (存储器总线, 或称为内存总线), 491, 498
- Memory chunks (内存块), 76
- Memory contents (内存内容), 60f
- Memory fragmentation (内存碎片化), 95
- Memory management (内存管理)
 - advanced memory management, types of (高级的内存管理, 类型), 248-252
 - defined (关于内存管理的解释), 21
 - fundamentals of (内存管理基础), 75-76
 - Linux (Linux 操作系统内存管理), 451-452
 - Palm (Palm 操作系统内存管理), 471-472
 - process and (进程和内存管理), 58-63
 - single-user multitasking OS and (单用户多任务操作系统和内存管理), 75-80
 - system (内存管理系统), 110
 - Windows NT (Windows NT 操作系统内存管理), 425-427
 - 见 Basic memory management
- Memory management unit (MMU, 内存管理部件), 102-103, 425, 446
- Memory manager (MM, 内存管理器, 或称为内存管理程序), 451
- Memory mapped files (内存映射文件), 190, 249-250, 292-293
- Memory pointer table (MPT, 内存指针表), 472
- Message (消息)
 - certificates (证书), 377
 - digest (消息摘要), 376-377, 376f
 - oriented attributes (面向消息的属性), 185, 187
 - passing (消息传递), 134, 184
 - queuing (消息队列), 189
 - signing (消息签名), 376f
- Message passing interface, parallel virtual machines (MPI/PVM, 消息传递接口和并行虚拟机), 134
- Meta-attributes (元属性), 431
- Methods (方法), 37
- Metrowerks (Metrowerks 公司), 476
- MFT record (主文件表记录), 429f, 430f
- Microelectromechanical systems (MEMS, 微机电系统), 325
- Microkernel system (微内核系统), 34, 35f, 117, 118f
- Microprogramming (微程序设计), 495-496
- Microsoft (微软公司)
 - DOS (微软 DOS 操作系统), 284
 - NFS (微软网络文件系统), 399
 - OS (微软操作系统), 394
 - Windows NT (微软 Windows NT 操作系统), 415
- Middleware (中间件), 134, 387-388
- Middleware service layer (中间件服务层), 387f
- Migration (迁移), 387
- Minimalist philosophy (最小化哲理), 16, 40

- Miniport drivers (微型端口驱动程序), 439
 - MINIX file system (MINIX 文件系统), 115, 452
 - MIPS (无锁管道阶段的处理器架构)
 - CPUs (MIPS 处理器), 446
 - R4000 (MIPS R4000 处理器), 248
 - RISC processor (MIPS RISC 处理器), 418
 - Mirroring (磁盘镜像), 309, 310
 - Mirror of stripes (条纹式单镜像), 312
 - Mkdir command (用于创建目录的命令 mkdir), 264
 - Mnemonic name (助记符), 491
 - Mobile IP (移动式网际协议), 395-396
 - Mobile networks (移动网络), 506
 - Mobile wearable devices (穿戴式移动设备), 484
 - Modal form (模态窗体), 83
 - Model (模型), 333-335
 - Modifier field (修饰符字段), 487
 - Modules (模块), 70
 - Monitor (监控程序; 管程), 48-49, 196-197
 - Monitor mode (监控模式), 29
 - Monitor program (监控程序), 48
 - Monolithic architecture (整体式体系结构), 25
 - Monolithic kernel (整体式内核, 或称为大内核), 33, 116
 - Monolithic single-kernel (整体式单内核), 33
 - Moore's law (摩尔定律), 128, 140
 - Most significant byte (MSB, 最高有效字节), 187
 - Motherboard (主板), 50
 - Motorola 68000 (摩托罗拉 68000 处理器), 91, 103, 109, 486
 - Motorola 68020 (摩托罗拉 68020 处理器), 101-103
 - Mounting (挂载), 288-290
 - Mouse device driver (鼠标驱动程序), 8
 - MPT chaining (主控指针表链接), 79f
 - MS-DOS (微软 DOS 操作系统), 104-105, 212-213, 452
 - MS-DOS partitions (微软 DOS 操作系统分区), 436
 - M68k processor (摩托罗拉公司的嵌入式 m68k 处理器), 446
 - MSRPC (微软远程过程调用), 461
 - Multicast (多点传送, 或称为多播或组播), 184, 186, 404
 - Multicast addresses (多播地址), 349
 - Multicast packets (组播式数据包), 335, 336
 - Multicore chips (多核芯片), 496
 - Multicore processor (MCP, 多核处理器), 124
 - MULTICS (MULTICS 操作系统), 31, 114, 236
 - MultiFinder (多重查找器), 98, 99-100, 99f
 - Multilevel indexed file (多级索引文件), 279f
 - Multilevel indexes (多级索引), 278-279
 - Multilevel page table (多级页表), 231, 426f
 - Multilevel queues (多级队列), 160
 - Multiple connections (多个连接, 或称之为多重连接), 185, 186
 - Multiple data streams (多个数据流, 或称之为多重数据流), 435-436
 - Multiple hardware (多种硬件), 421-423
 - Multiple interactive users (多个交互式用户, 即多用户交互), 30
 - Multiple page size (页面大小多样化), 233
 - Multiple processes (多进程), 218-220f, 218-222
 - Multiple threads (多线程), 168f
 - Multiple tiers (多层), 505
 - Multiple-user OS (多用户操作系统), 113-125
 - environment of (多用户操作系统环境), 121-123, 123f
 - historical overview of (多用户操作系统的历史), 114-116
 - purpose of (多用户操作系统的目的), 113-114
 - Multiple users (多个用户或多用户), 107-108
 - Multiprocessing (MP, 多处理), 48, 128-129, 132
 - Multipurpose Internet mail extensions (MIME, 多用途互联网邮件扩充协议), 340
 - Multitasking (多任务), 63, 69-71, 69f, 97f
 - Multitasking OS (多任务操作系统), 30
 - Multitasking system design (多任务系统设计), 70
 - Multithreaded server (多线程服务器), 392f
 - Multithreading (多线程), 132
 - Musical instrument device interface (MIDI, 乐器设备接口), 472
 - Mutex (互斥机制; 二值型信号量, 即互斥信号量), 192, 196
 - Mutual exclusion (互斥条件; 互斥), 201-202, 401-402, 401f, 402f
- N
- Named pipes (命名管道), 189
 - Name resolution (名称解析), 343
 - Name servers (域名服务器), 505
 - Name space (名称空间), 394

- Naming (命名), 394-396
- Naming strategy (命名策略), 184, 186
- Nanokernel (超微内核), 103
- National Computer Security Center (NCSC, 国家计算机安全中心), 380
- Native command queuing (NCQ, 本地命令式排队), 321
- N buffer (N 个缓冲区), 185
- NEC v850 (NEC 公司的 v850 处理器), 446
- Negative acknowledgment (NACK, 否定应答), 405, 405f
- NetBEUI (网络基本输入/输出系统增强型用户接口协议), 440, 456
- NetBIOS (网络基本输入/输出系统协议), 345, 456, 461
- .net remoting (.net 远程处理), 134, 135
- Netware (Netware 系统服务器), 345
- Network (网络)
- computer system, use of (计算机系统, 网络的使用), 504-506, 506f
 - functional classes of (网络功能类别), 31-32
 - Linux, use of (Linux 操作系统, 网络的使用), 460-462
 - troubleshooting (网络故障排查), 354
 - types of (网络分类), 506
 - Windows NT, use of (Windows NT 操作系统, 网络的使用), 440-441
- Network access (网络访问), 24
- Network adapter (网络适配器), 346
- Network address translation (NAT, 网络地址转换), 344
- Network attached storage (NAS, 网络附属存储器), 129
- Network controller (网络控制器), 505
- Network devices (网络设备), 299, 455-457
- Network driver interface specification (NDIS, 网络驱动程序接口规范), 440
- Network file system (NFS, 网络文件系统), 289-292, 292f, 387, 399
- Network General Corporation (网络通用公司), 355
- Network interface (NET, 操作系统的网络接口模块), 460
- Network interface card (NIC, 网络接口卡, 简称网卡), 346, 349, 460
- Network interface controller (NIC, 网络接口控制器), 505
- Network I/O devices (网络型输入/输出设备), 502
- Network layering (网络分层), 460-461, 461f
- Network management (网络管理), 354-355
- Network Neighborhood protocols (网上邻居协议), 461
- Network News (网络新闻), 399
- Network Object Model Environment (网络对象模型环境), 459
- Network programming (网络编程), 473-474, 475t
- Network protection (网络保护), 378-380, 379f
- New state (新状态), 27, 154-155
- New Technology (NT, 新技术, 即 NT 操作系统名称的寓意), 418
- NeXT Computer Inc. (NeXT 计算机公司), 38, 107, 110
- Next fit (循环首次适应算法), 220
- NextStep OS (NextStep 操作系统), 38, 107
- Nice command (一条用来改变程序优先级的命令 nice), 449-450, 450
- Nmap (一款端口扫描程序, 指网络映射器), 462
- No buffer (没有缓冲区), 185
- Nodes (进程状态转换图中的结点; 集群中的节点; 处理器节点), 26, 129, 145
- No guarantee semantics (不保证性语义), 408
- Nonblocking read (非阻塞式读取), 188
- Non-moveable chunks (不可移动的内存块), 78
- Nonresident attributes (非常驻属性), 429-430
- Non-Uniform Memory Access (NUMA, 非均匀性内存访问), 129
- Nonvolatile memory (非易失性存储器), 499
- Noop scheduler (Noop 调度器), 315, 458
- Normal process (普通进程), 449
- Notebook computer (笔记本计算机, 或称为笔记本电脑), 484
- Notepad (一款文本编辑器程序名称, 寓意记事本), 92, 264
- Not recently used (NRU, 最近未使用算法), 244
- Not used recently (NUR, 最近未使用算法), 244
- Novell Directory Service (NDS, Novell 目录服务), 395
- Novell Netware (诺勒公司网络操作系统), 184, 440
- Nonvolatile random access memory (NVRAM, 非

- 易失性随机访问存储器), 75-76
- N-step SCAN (N 轮向前看调度算法), 320
- N10 (N-Ten) (处理器芯片名; 微软所用 i860 仿真器名称), 418
- NT file system (NTFS, NTFS 文件系统, 或称为 NT 文件系统), 428-433
- advanced features of (NTFS 文件系统高级功能特征), 433-436
- goals of (NTFS 文件系统的目标), 431-433
- N-tier applications (N 层应用程序), 389-390, 390f
- NTLM (NT 局域网管理器), 462
- NT-style printing service (SPOOLSS, NT- 风格打印服务), 462
- NuBus (苹果系统总线名称, 寓意新型总线), 104
- Null pointer catcher (空指针捕获器), 425
- Number of records (记录数), 57
- ## O
- Object (对象), 37, 81, 83
- Object adapter (对象适配器), 398
- Object code (目标代码), 491
- Object linking and embedding (OLE, 对象链接和嵌入), 434
- Object Management Group (OMG, 对象管理组), 398
- Object module (目标模块), 210
- Object-oriented approach (面向对象的方法), 37-38
- Object-oriented architecture (面向对象的体系结构), 25
- Object programs (目标程序), 491
- Object request broker (ORB, 对象请求代理), 398
- Offline storage (离线存储器, 或称为脱机存储器), 499
- Offset (偏移量), 487
- One buffer (一个缓冲区, 或称为单缓冲), 185
- One-to-one mapping (一对一映射), 170
- One-to-one thread mapping model (一对一线程映射模型), 170f
- One-way connections (单一连接), 184, 185
- Online storage (在线存储器, 或称为联机存储器), 499
- Opcode (operation code, 操作码字段), 486
- Open files (打开的文件), 26
- Open files table (打开文件表), 37
- OpenGL (一套三维图形处理库), 109
- Open Group (开放集团), 398
- Open shortest path first (OSPF, 开放式最短路径优先协议), 342
- Open source implementations (开源实现), 465t
- Open source projects (“开源”项目), 115
- Open Source Samba (开源 Samba 软件包), 345
- Open source systems (开源系统), 8
- Open standards (开放标准), 387
- Operand fields (操作数字段), 487
- Operating system (OS, 操作系统), 3-17
- basic functionality's of (操作系统基本功能), 5-6
- basic terminology of (操作系统基本术语), 10-11
- concepts of (操作系统的概念), 138-142
- functions of (操作系统的功能), 20-25, 25f
- historical overview of (操作系统发展概述), 15-17, 16f
- images of (操作系统视图), 11-13, 12f, 13f
- 360 TSO (带有分时选项的 360 操作系统), 31
- 2 (OS/2 操作系统), 295, 417, 422, 428, 452
- Optimal page replacement (OPT, 最佳页面淘汰算法), 241
- ORG 100 (一条将模块放在位置 100 的汇编指令), 211, 212
- ORG 500 (一条将模块放在位置 500 的汇编指令), 211
- Orphan process (孤儿进程), 165
- OSI model (开放系统互连参考模型, 或称为开放系统互连模型), 334, 334f
- Output devices (输出设备), 484-485
- Overlay (覆盖), 48, 51, 62
- Overlay memory management (覆盖式内存管理), 61-63, 62f
- Overlay single process (支持覆盖的单一进程), 215, 215f
- Overloading (重载), 288
- ## P
- Packet capture (数据包捕获), 355
- Page access (页面访问), 230-231
- Page access protection (页面访问保护), 227f, 230-231
- Page directory (页目录表), 426
- Paged memory access (基于分页的内存访问), 227f
- Page fault (缺页故障), 102, 239, 427

- Page fault frequency (PFF, 页面故障频率), 245
- Page locking (页面锁定), 246-247
- Page mapping (页面映射), 227f, 426, 426f
- Page reference bit (页面引用位), 242
- Page replacement (页面替换, 或称为页面置换或页面淘汰), 240, 427
- Page sharing (分页共享), 426-427
- Pages per process (每进程页框), 244-245
- Page table (页表), 37, 233, 501
- Page table address register (页表地址寄存器), 227
- Page table demand paging (请求分页式页表), 239f
- Page table length register (页表长度寄存器), 230
- Paging (分页), 226-233
- PAI (个人应用集成), 480
- Palm, Inc. (Palm 公司), 67, 69
- Palm OS (Palm 操作系统), 71-73, 72f, 469-480
- Palm Pilot (Palm Pilot 手持式掌上电脑), 69f
- Palm Powered™ (Palm Powered 手持式设备), 474
- PalmSource, Inc. (PalmSource 公司), 70, 480
- Palm Sync (Palm 系统的一款应用程序, 用于实现手持式装置上的数据库文件与个人计算机上的数据库文件之间的同步), 74
- Pande Group (潘德小组), 144
- Parallel computing (并行计算), 128
- Parallel connection (并行连接), 504
- Parallelism (并行), 128
- Parallel Line Internet Protocol (PLIP, 并行线路互联网协议), 457
- Parallel processing (并行处理), 128-132, 130f, 131f
- Parent process (父进程), 123, 164-165
- PA-RISC (惠普公司的 PA-RISC 处理器), 446
- Parity schemes (奇偶校验方案), 312f
- Partially connected mesh (网状网络拓扑结构), 337f
- Partitioning (分区), 306, 306f
- Partitions (磁盘分区), 305-306, 436-437
- Partition table (分区表), 307
- Pascal (Pascal 程序设计语言), 39, 476
- Passwords (密码), 108, 367
- Patch (修补, 即打补丁), 211
- Path (路径名或路径), 261, 394
- PC (个人计算机)
- Card (个人计算机卡, 或称为 PC 卡), 104, 353, 438, 446
 - characteristics of (个人计算机的特征), 50-52, 51f
 - Exchange (7.1 版 Mac 操作系统上的一款能够访问 MS-DOS 格式软盘的软件, 寓意“个人计算机交换”), 104
 - MCIA (插槽类型名称, 寓意个人计算机存储卡国际协会), 104-105, 438
 - OS (个人计算机操作系统), 90
 - relative addressing (程序计数器相对寻址), 487
- PDP-7 (一款小型计算机), 114
- Peer-to-peer (P2P, 点对点系统; 对等模型), 130, 504
- Pentium (奔腾处理器), 137
- Performance (性能), 182, 386
- Performa 6100 (一款 Mac 计算机型号), 103
- Peripheral Component Interconnect (PCI, 外围部件互连), 104
- Peripheral devices (外围设备), 502
- Per-process open file table (进程打开文件表), 122
- Persistence (持久性), 188, 387
- Persistent attributes (持久通信属性), 185
- Persistent data (持久性数据), 76
- Personal area networks (PANs, 个人区域网络), 353
- Personal data interchange (PDI, 个人数据交换), 474
- Personal data synchronization (个人数据同步), 473-474
- Personal desktop computer (个人台式计算机), 484
- Personal digital assistant (PDA, 个人数字助理, 或称为掌上电脑), 4, 68, 353, 473
- Personal identification number (PIN, 个人身份号码), 367
- Personal information managers (PIMs, 个人信息管理器), 68
- Personal Software (个人软件公司), 417
- Personal Web Sharing (个人网页共享), 105
- Per-user disk space quotas (每用户磁盘空间配额), 434
- Phishing (网络钓鱼), 367
- Physical address extension (PAE, 物理地址扩展), 102, 214, 446
- Physical disk organization (磁盘物理组织), 302-305, 302f, 305t
- Physical layer (物理层), 334, 352-354
- Physical location of information (信息的物理位置), 183, 386

- Physical memory space (物理内存空间), 132
- Physical space (物理空间), 214
- Physical structure (物理结构), 262
- Physical virtual machine (物理虚拟机), 392-393, 393f
- Pickup (搭便车调度算法), 315, 316f
- Ping (一种用于验证两台设备之间连接的实用工具例程), 354
- Ping of Death (死乒攻击), 363
- PINGs (“拼接”测试), 345
- Pipeline architectures (流水线体系结构), 194
- Pipeline flow graph (管道流图), 131f
- Pipelining (流水线), 494
- Pipes (管道), 188
- Plain old telephone service (POTS, 普通老式电话服务), 350, 351
- Plaintext (消息明文), 374
- Platform agnostic (平台无关的), 134
- Plug and play (PnP, 即插即用), 437-438
- Pocket PC (掌上电脑), 68
- Pointing device (指针式设备), 8, 17
- Point-to-Point Protocol (PPP, 点对点协议), 457
- Polling (轮询), 300
- POP3 (post office protocol version 3, 第三版邮局协议), 340-341, 344, 461
- Port (端口), 189, 439
- Port 21 (21 号端口), 340
- Port 80 (80 号端口), 339, 340
- Portability (可移植性), 49, 52
- Portable batch system (PBS, 轻便型批处理系统), 129, 145-146
- Portals (门户), 146-147
- Port mirroring (端口镜像), 355
- Port number (端口号), 338
- Port scanners (端口扫描程序), 462
- POSIX (UNIX 可移植操作系统接口), 115, 173-174
- Power Macintosh (强力 Mac 计算机), 103, 104
- Power-On Self-Test (POST, 开机自检), 96
- PowerPC (一款处理器, 寓意强力个人计算机), 103, 110, 418
- Practical network layer model (实用的网络层次模型), 334f
- PRC-Tools (一款基于 gcc 的采用 C/C++ 语言来构建 Palm 操作系统应用程序的编译器工具链), 476
- Preallocation (预分配), 270
- Preemption (抢占), 75, 159, 205-206
- Preemptive multitasking (抢占式多任务), 124, 110
- Prefetch file (预提取文件), 251
- Prefetch profiles (预取配置), 427
- Pretty good privacy (PGP, 恰到好处隐私协议), 378
- Prevention (死锁预防), 201-203
- Primary data (主数据), 267
- Primary key (主键), 267
- Primary memory, management of (主内存或称为主存或内存, 管理), 209-210
- Primary partitions (主分区), 306
- Primary rate interface (PRI, 主速率接口), 351
- Primary storage (主存储器), 70, 496-497, 498
- Printer queue (打印机队列), 37
- Priority inversion (优先级倒置), 195
- Priority scheduling (优先级调度), 157
- Priority with preemption (抢占式优先级), 159
- Private key (私钥), 375
- Privileged mode (特权模式), 29, 490
- Problems and threats (问题与威胁). 见 Security and protection
- Process (进程)
- concept of (进程的概念), 25-27
 - defined (关于进程的解释), 20, 151
 - types of (进程分类), 28-29
- Process control block (PCB, 进程控制块), 28, 28f, 152-153, 153f
- Process descriptor (进程描述符), 152
- Processes (进程), 28-29
- Process groups (进程组), 407
- Process identifier (ID, 进程标识符), 28
- Processing graffiti input (处理涂鸦式输入), 73
- Processing node (处理节点), 131
- Process management (进程管理), 21, 151-178
- Processor (处理器), 485-496
- Processor affinity (处理器亲和性), 133, 164
- Processor cache (处理器高速缓存), 491
- Processor chip (处理器芯片), 496
- Processor control (处理器控制), 491-492
- Processor data path (处理器数据通路), 491-492

- Processor instruction execution cycle (处理器指令执行周期), 493-494
- Processor instruction set architecture (处理器指令集体系结构), 486-490
- Processor interrupts (处理器中断), 494-495
- Processor machine language (处理器机器语言), 486-490, 488f, 489f
- Processor microprogramming (处理器微程序设计), 495
- Processor multicore chips (处理器多核芯片), 496
- Processor pipelining (处理器流水线), 493-494
- Process resilience (进程韧性), 407-408, 407f, 408f
- Process scheduler (进程调度器, 或称为进程调度程序), 27, 75
- Process Scheduler module (SCHED, 进程调度模块), 447
- Process scheduling (进程调度)
- Linux (Linux 操作系统进程调度), 447-450
 - single-user multitasking OS, use of (单用户多任务操作系统, 进程调度的使用), 73-75
 - Windows NT (Windows NT 操作系统进程调度), 423-424
- Process sharing (共享进程中代码和数据), 168f
- Process state information (进程状态信息), 153
- Process states (进程状态), 25-27, 26f, 152, 154
- Process synchronization (进程同步), 138-140
- Process threads (进程线程), 123-125, 123t, 125f, 172-173
- Proc file system (进程文件系统 /proc), 454
- Producer-consumer problem (生产者-消费者问题), 195-196
- Program binary (二进制程序; 程序二进制文件), 59, 491
- Program counter (程序计数器), 25, 152, 492
- Program header (程序头), 61
- Program image (程序映像), 59
- Programmable read-only memory (PROM, 可编程只读存储器), 71, 75-76
- Programmer fragmentation (程序员碎片), 274, 275
- Programming (程序设计, 或编程), 439, 458-460
- Programming environments (编程环境), 475-476
- Program processor (程序处理器), 491
- Programs (程序), 491
- Program status registers (程序状态寄存器), 492
- Progress dialog (进度对话框), 83
- Promiscuous mode (混杂模式), 355
- Proportional allocation (按比例分配), 245
- Protection and security (保护与安全). 见 Security and protection
- Pseudo file system (虚文件系统, 或称为伪文件系统), 454
- Psion (佩森公司), 476
- Pthreads (一项与线程有关的 POSIX 标准), 173, 174
- Public key (公钥), 375
- Purchased components (组件购买), 182, 386
- ### Q
- Quality of service (QoS, 服务质量), 348
- Quantum (时间片, 即额定时间量), 447
- QuickTime (一款多媒体软件, 寓意快乐时光), 102
- ### R
- Race condition (竞态条件), 138, 191
- Race hazard (竞争险象), 191
- RAID (redundant array of independent disks, 廉价磁盘冗余阵列), 309-314
- RAID failure (廉价磁盘冗余阵列故障), 313-314
- RAM disk driver (内存型磁盘驱动程序, 或称为 RAM 磁盘驱动程序或随机存取存储器型磁盘驱动程序), 473
- RAM management (内存管理, 或称为随机存取存储器管理), 100
- RAM methods (随机存取存储器方法), 266-267, 266f
- RAM sequential access (随机存取存储器顺序访问), 298
- Random access (随机存取, 或称为随机访问), 266-267, 266f
- Random access memory (RAM, 随机访问存储器, 或称为随机存取存储器), 50, 75-76
- Raw access (原始存取), 268-269
- Raw access method (原始存取方法), 268-269
- Raw interface (原生接口), 82
- Raw I/O (原始输入/输出), 299, 306
- Read message (读取消息), 373
- Read next operation (读取下一条记录的操作命令), 266
- Read-only memory (ROM, 只读存储器), 6, 14,

- 48, 71, 75-76, 96
- Read-only support (只读支持), 433
- Read-the-button software (一款名为 read-the-button 的软件), 5
- ReadyBoost (闪速缓存机制), 420
- Ready process queue (就绪进程队列), 22-23, 37, 156
- Ready state (就绪状态), 27, 155, 156
- Ready to run (就绪状态), 26
- Real-time application interface (RTAI, 实时应用程序接口), 464
- Real-time functional classes (实时功能类型), 32-33
- Real-time Linux common API (实时 Linux 通用应用程序接口), 463-464
- Real-time OS (RTOS, 实时操作系统), 464
- Real-time processes (实时进程), 448-449
- Real-time tasks (实时任务), 471
- Real-time threads (实时线程), 424
- Records (记录), 80-81
- Recoverability (可恢复性), 431, 431-432
- Redirection (重定向), 290-292, 291f, 292f, 437
- Redirector (重定向器), 292
- Reduced Instruction Set Computer (RISC, 精简指令集计算机), 418, 486
- Reed-Solomon codes (里德-所罗门码), 308
- Reentrancy (重入), 172
- Reference count (引用计数), 242
- Regions (区域), 236
- Register (寄存器), 491-492, 497
- Register addressing (寄存器寻址), 487
- Registration functions (注册函数), 119t
- ReiserFS (一种新型 Linux 文件系统), 295
- Relative pathname (相对路径名), 263
- Release number (版本号), 116
- Reliability (可靠性), 183, 386
- Reliable client-server Communication (可靠的客户端-服务器通信), 408
- Reliable multicast communication (可靠的多播通信), 404-405, 405f
- Reliable sleep state (可靠的睡眠状态), 420
- Relocation (重定位), 387
- Relocation hardware (重定位硬件), 213, 213f
- Remainder section (剩余区), 192
- Remote file systems (远程文件系统), 289-290
- Remote method invocation (RMI, 远程方法调用), 134, 141
- Remote monitoring (RMON, 远程监控), 355
- Remote procedure call (RPC, 远程过程调用), 141, 187, 392, 396-398, 397f
- Remote services, use of (远程服务, 运用), 141-142
- Removable discs (可移动磁盘), 498-499
- Removable medium (可移动介质), 371
- Reparse point (重解析点), 435
- Replacement algorithms (淘汰算法, 或称为置换算法), 243-244
- Replicated database (复制型数据库), 391
- Replication (复制), 387
- Replication transparency (复制透明性), 32
- Repudiate (否认), 377
- Request for comments (RFC, 互联网标准草案), 335-337
- Reserved area (保留区), 55
- Resident attributes (常驻属性), 428-429
- Resource-allocation graph (资源分配图), 200, 200f
- Resource management, types of (资源管理, 类型), 22-24
- Resource processes sharing (进程共享资源), 198f
- Resources (资源), 81
- Resource utilization (资源利用率), 132
- Ring topologies (环形网络拓扑结构), 337f
- Ritchie, Dennis (丹尼斯·里奇), 114
- Rivest, Shamir, Adleman (RSA, 一种公开密钥加密标准算法), 376
- Rm command (一条用来删除文件的命令 rm), 264
- Rmdir command (一条用来删除目录的命令 rmdir), 264
- Robots (机械人, 即僵尸机器), 363
- Roles (角色), 369-370
- Roll-out/roll-in (滚出/滚入技术), 216
- Rotational latency (旋转延迟时间), 304-305
- Round robin (轮转策略), 159, 449
- Routers (路由器), 342, 353, 505
- Routing information protocol (RIP, 路由信息协议), 342
- RPC stub (远程过程调用的桩例程), 397f
- RSA protocol (RSA 协议), 375
- RTLinux (一种基于 Linux 操作系统内核的实时系

- 统模型), 464
- Run (运行), 210
- Runnable program (可运行程序), 59
- Running (运行的), 26
- Running state (运行状态), 27
- Runqueue (运行队列), 447, 448
- S**
- Safe state (安全状态), 204f
- SAINT (一款知名的端口扫描软件), 462
- SAMBA (一套在 UNIX 和 Linux 操作系统中用来访问微软服务器的软件包), 461-462
- Sandbox (沙箱), 365
- Sandbox execution model (沙箱执行模型), 365f
- SATAN (一款知名的端口扫描软件, 寓意网络分析安全管理工具), 462
- Scaling (规模扩展, 或称为伸缩性), 182, 386
- SCAN (扫描算法), 318
- Scandisk (磁盘检查修复工具例程), 293
- Sched_setparam (一个用来更改进程自身优先级的操作系统函数), 450
- Scheduling, process (调度, 进程), 156-164
- Scientific computing (科学计算), 142-143
- Screen savers (屏幕保护程序), 332
- Screen size (屏幕尺寸), 72
- Screen window (屏幕窗口), 74
- Scripts (脚本), 59, 365-366, 365f
- SDLC (同步数据链路控制协议), 184
- Search for Extra-Terrestrial Intelligence (SETI, 外星文明探寻项目), 175, 183
- Secondary key (次键), 267
- Secondary storage (辅助存储器, 或称为第二级存储器), 23, 71, 81, 498
- Second chance algorithm (二次机会算法), 243
- Secret key (隐秘密钥), 375
- Sector addressing (扇区编址), 303-304
- Sector count zones (扇区分组区), 303-304
- Sector relocation (扇区迁移, 或称为扇区重定位), 323-324
- Sectors (扇区), 54, 302-303, 302f
- Sector sparing (扇区保留), 323-324
- Secure Accounts Manager (SAM, 安全账户管理器), 461
- Secure hash standard (SHA, 安全散列标准算法), 377
- Secure HTTP protocol (安全超文本传输协议), 378
- Secure socket layer (SSL, 安全套接字层协议), 378
- Security administration (安全管理), 380
- Security and protection (安全与保护), 359-381
 - policies for (关于安全与保护的策略), 370-373
 - problems concerning (关于安全与保护的问题), 360-366
 - techniques for (关于安全与保护的技术), 370-373
 - threats concerning (关于安全与保护的威胁), 360-366
- Security descriptor (安全描述符), 430
- Security-Enhanced Linux (SELinux, Linux 安全增强模块), 462
- Security protocols (安全协议), 377-378
- Seek (寻道), 304-305, 305t
- Seek time (寻道时间), 304
- Segmentation (分段), 233-236
- Segmentation with paging (段页式), 236-238, 237f
- Segmenting a process (进程分段), 234f
- Segment number (s, 段号), 234
- Self-monitoring and reporting technologies (S.M.A.R.T., 自我监控报告技术), 324
- Semaphore (信号量), 193
- Send message (发送消息), 373
- Sequential access (顺序存取, 或称为顺序访问), 265-267, 265f
- Sequential file with current record pointer (顺序文件及当前记录指针), 265f
- Serial connection (串行连接), 504
- Serial Line Interface Protocol (SLIP, 串行线路接口协议), 457
- Server computers (服务器计算机), 484
- Server message block (SMB, 服务器消息块协议), 345, 399, 461
- Servers (服务器), 31, 32
- Services (服务), 11
- Service security improvements (服务安全提升), 419
- Service shutdown ordering (服务关闭排序机制), 420
- Service variability (服务可变性), 319
- SETI@Home (外星文明探寻之家), 130, 137, 144
- SETI project (外星文明探寻项目), 332
- SHA-1 (一种用来生成散列码的算法), 378

- Shared key (共享密钥), 375
- Shared memory (共享存储器, 或称为共享内存), 190, 192f
among processes (进程间共享内存), 248-249
systems (共享内存系统, 或称为共享存储器系统), 190
- Shared variable (共享变量), 191f
- Shell (外壳程序, 即命令解释器), 60
- Shell command (外壳程序命令, 或称为 shell 命令), 166
- Shell interpreter (shell 命令解释器, 即外壳程序), 11
- Shielded twisted pair (STP) wiring (屏蔽型双绞线布线方式), 352
- Shift (Shift 按键), 6
- Shmid (共享内存标识符), 249
- Shortest job next (SJN, 最短作业优先调度), 158
- Shortest positioning time first (SPTF, 最短定位时间优先调度算法), 316-317, 316f
- Shortest remaining time first (SRTF, 最短剩余时间优先调度), 158-159
- Shortest runtime first (SRTF, 最短运行时间优先调度), 158
- Shortest seek time first (SSTF, 最短寻道时间优先调度算法), 316
- Short message service (SMS, 短信服务), 474, 479
- Short-term scheduler (短程调度器), 156
- Signal system call (signal 系统调用), 193
- Signal routine (signal 例程), 194
- Signatures (签名), 379
- Silicon Graphics (Silicon Graphics 计算机系统), 248, 418
- Simple mail transfer protocol (SMTP, 简单邮件传输协议), 340-341, 344
- Simple Network Management Protocol (SNMP, 简单网络管理协议), 354-355, 441
- Simultaneous multithreading (SMT, 同时多线程), 172
- Single connections (单一连接), 185, 186
- Single instance storage (SIS, 单实例存储), 434
- Single-level directory (单级目录), 259-260, 260f
- Single process of memory management (单一进程内存管理), 211-216
- Single-process OS (单进程操作系统), 47-63
- Single tasking (单任务), 92-93, 93f
- Single UNIX Specification (SUS, 单一 UNIX 规范), 115
- Single user (单一用户), 30
- Single-user multitasking/multithreading OS (单用户多任务 / 多线程操作系统), 89-110
historical overview of (单用户多任务 / 多线程操作系统发展概述), 89-90
origin of (单用户多任务 / 多线程操作系统的起源), 90-91
- Single-user multitasking OS (单用户多任务操作系统), 67-85
- Single-user single-tasking OS (单用户单任务操作系统), 29
- Single-user systems (单用户系统), 15
- Skeleton (程序框架), 398
- Slab allocator (内存对象集分配器, 或称为 Slab 分配器), 451
- Sleep mode (睡眠模式), 71
- Sliding window (滑动窗口), 240
- Small computer system interface (SCSI, 小型计算机系统接口), 118, 439
- Small real-time OS (小型实时操作系统), 464
- Small records, blocking of (针对短小记录的分块技术), 301-302
- Small systems device families (小微系统设备系列), 479t
- Smalltalk (Smalltalk 编程语言), 476
- S.M.A.R.T. (自我监控报告技术), 324
- SMP load balancing (对称多处理负载均衡), 450
- SMP systems (对称多处理系统), 194-195
- SNA (系统网络体系结构), 184, 344, 456
- Sniffer (一种用于捕获数据包的嗅探器软件), 355
- Snooping (窥探), 195
- Socket (套接字), 189, 456
- Socket buffer (skbuff, 套接字缓冲区), 460
- Soft deadlines (软性的截止时间), 32
- Software, OS (软件, 操作系统), 12f
- Software Development Kit (SDK, 软件开发工具), 476
- Software events (软件事件), 495
- Software tools (软件工具), 132
- Solaris (Solaris 操作系统), 33, 110, 175-176, 249
- Solutions, real-world (实用的解决方案), 205-206

- Sony (索尼公司), 69
- Source code (源代码), 491
- Source programs (源程序), 491
- Source route bridging (源路由桥接), 348
- Space tracking (空间监测), 431
- SPAM email (垃圾邮件), 345
- Spanning tree (生成树), 347
- SPARC (斯巴克处理器), 446
- Sparse files (稀疏文件), 434-435
- Spatial locality (空间局部性), 500
- "Spawning a child" (产生一个子进程), 165
- Special forms, types of (特殊的窗体类型), 83-84
- Special memory management (特殊的内存管理), 249-251, 252f
- Special-purpose registers (专用寄存器), 492
- Speed of light (光速), 128
- Speeds (速度), 304-305, 305t
- Spin-lock (自旋锁), 193
- SPOOLING system (假脱机系统), 157
- Spyware (间谍软件), 362-363
- Stack (堆栈, 或简称栈), 61
- Stack sniffer (栈嗅探器), 95
- Stack space (栈空间), 166
- Standard file systems (标准文件系统), 452
- Standard information (标准信息), 430
- Star topologies (星形网络拓扑结构), 337f
- Startup speed of XP (XP 操作系统启动速度), 441-442
- Starvation (饥饿问题), 157
- Stateless (无状态的), 366
- States (状态), 28, 154-156, 154f, 494
- State transitions (状态转换), 26
- Static data (静态数据), 59, 61
- Static IP address (静态的网际协议地址), 343
- Static random access memory (SRAM, 静态随机存取存储器), 497
- Status-driven system (状态驱动的系统), 36
- Stderr (standard error, 标准错误输出接口), 82
- Stdin (standard input, 标准输入接口), 82
- Stdin/stdout function (标准输入/输出接口函数), 472
- Stdout (standard output, 标准输出接口), 82
- Stealth port scanners (隐式端口扫描器), 462
- Storage area network (SAN, 存储域网络), 129
- Storage class (存储类), 439
- Storage driver (存储类驱动程序), 439
- Storage hierarchy (存储体系, 或称为存储器层次结构), 484, 497-500, 497f, 501f
- Storage I/O devices (存储型输入/输出设备), 502
- Storage port (存储端口), 439
- Storage random access memory (存储型随机访问存储器), 76
- Storage units (存储单位), 496-497
- Storage volatility (存储易失性), 498
- Streaming (流), 185, 187
- Stream I/O (流输入/输出), 472
- Stream of data (数据流), 187
- Strict priority mechanism (严格的优先级调度机制), 160
- Strip (条纹划分), 309
- Striped disk array (条纹式磁盘阵列), 309-310
- Stripe of mirrors (条纹式多镜像), 312-313, 312f, 313f
- Striping (条纹划分), 309
- Strong passwords (高强度密码), 367
- Stylus tracking (手写笔跟踪), 73
- Subfiles (CP/M 系统中的批处理文件类型的名称), 59
- Subroutines (子程序, 或称为子例程), 61
- Subsystems (子系统), 422
- Suites (套件), 341
- Summary Information (摘要信息), 436
- Sun Microsystems (太阳微系统公司), 110, 291, 393, 454
- SuperFetch (超级预提取机制), 420
- Supervisor mode (管理程序模式, 或称为管态), 29, 490
- Support models, threads (支持模型, 线程), 171f
- Suspended state (暂停状态), 155
- Swap file (对换文件), 243
- Swap instruction (一条基于对调变量取值原子操作的硬件锁指令), 193
- Swap out (对换到外存), 23
- Swapping (对换技术), 215-216, 216f
- Sweep workflows (横扫型工作流), 133
- Switch (交换机), 347
- Switcher (切换器程序), 97
- "Switcher" memory layout ("切换器" 内存布局),

- 97f
- Switches/switching (交换机/交换), 334, 336, 346-347
- Switching devices (交换设备), 505
- Syllable(一个关于面向对象操作系统的研究项目), 38
- Symbian™ (塞班公司; 塞班操作系统), 68, 251, 252f, 452, 476
- Symbolic Optimizing Assembler Program (SOAP, 符号优化式汇编程序), 211
- Symbolic references (符号引用, 或称为符号式引用), 262
- Symbol table (符号表), 119
- Symmetric algorithm (对称算法), 375
- Symmetric key encryption (对称密钥加密), 375, 375f
- Symmetric multiprocessing (SMP, 对称多处理), 125, 125f, 132-134, 133f
- Sync application (一款用于实现手持式装置上的数据库文件与个人计算机上的数据库文件之间的同步的应用程序), 74
- Synchronization (同步), 190-197, 400-406
- Synchronous attributes (同步通信属性), 185, 187-188
- Synchronous I/O (同步输入/输出), 187-188
- SYN Flood (洪泛攻击), 363
- Sysadmin (系统管理员), 121
- Sysgen (system generation, 一条用于生成系统的命令), 437
- System 1 (第1版 Mac 操作系统), 94f
- System 2 (第2版 Mac 操作系统), 96-97, 97f
- System 3 (第3版 Mac 操作系统), 98
- System 4 (第4版 Mac 操作系统), 98-100, 99f
- System 5 (第5版 Mac 操作系统), 98, 100
- System 6 (第6版 Mac 操作系统), 101
- System 7 (第7版 Mac 操作系统), 101-105
- System 8 (第8版 Mac 操作系统), 106-107
- System 9 (第9版 Mac 操作系统), 107-109
- System administrator (系统管理员), 7f, 8, 121
- System bus (系统总线), 491
- System calls (系统调用), 7, 36, 495
- System clock (系统时钟), 493
- System heap (系统堆), 95-96
- System idle task (系统空闲任务), 175
- System managers (系统管理员), 7t
- System partition (系统分区), 94
- System programmers (系统程序员), 7, 7f
- System program (系统程序), 21
- System timing (系统时序), 493
- System view (系统视图), 6, 8-10, 9f, 10f
- Systemwide open file table (系统范围的打开文件表, 简称系统打开文件表), 122
- System X (第10版 Mac 操作系统, 即 X 版 Mac 操作系统), 107
- ## T
- Tables (表), 37
- Tagged queuing (标记式排队), 321-322
- TAJ (一个关于面向对象操作系统的研究项目), 38
- Tanenbaum, Andrew (安德鲁·塔嫩鲍姆), 115
- Tape drives (磁带驱动器), 325
- Tapes (磁带), 499
- Task automation (任务自动化), 102
- Tasklet (小任务), 120
- Tasks (任务), 26, 151, 176
- TCP communication (支持传输控制协议的通信), 474
- TCP header format (传输控制协议报头格式), 339f
- TCP/IP (传输控制协议/网际协议), 341-345, 404, 440
- TCP/IP protocol suite(传输控制协议/网际协议簇), 334
- TCP Wrapper (tcpd, 传输控制协议包装器, 或称为传输控制协议包装程序), 463
- TDM circuits (时分多路复用电路), 350-351
- Telecommunications Industry Association (TIA, 美国通信工业协会), 352
- Telecommuting (远程办公), 333
- Telephony applications (电话应用程序), 475, 475t
- Teletype (电传打字机), 49
- Telnet (一款用于登录到远程机器的服务例程; 远程终端协议), 338, 344, 379, 461
- Temporal locality (时间局部性), 500
- Terminated state (终止状态), 27
- Tertiary storage (第三级存储器), 499
- Test and Set instruction (一种基于测试-设置原子操作的硬件锁指令), 192-193
- THE OS (THE 操作系统), 204

- Thin client (瘦客户端), 388
- Third-party service (第三方服务), 183, 386
- 32-bit clean (彻底 32 位的, 或称为纯粹 32 位的), 101
- Thompson, Ken (肯·汤普逊), 114
- Thrashing (抖动, 或称为颠簸), 245-246
- Thread control block (TCB, 线程控制块), 166
- “Thread of execution” (执行的线程), 166
- Thread pool (线程池), 171
- Thread priority relationships (线程优先级关系), 424f
- Threads (线程), 167f, 168f, 170-177, 170f, 171f, 174f, 391-392
- Thread-safe (线程安全的), 172
- 3Com (3Com 公司), 441
- 370 TFLOPS (每秒 370 万亿次浮点运算操作), 144
- Three-layer model (三层模型), 388-389, 389f
- Three-state model (三状态模型), 154
- Throughput (吞吐量), 132, 160
- Ticket granting agencies (票据授予机构), 135
- Time division multiplexing (TDM, 时分多路复用), 350
- Time quantum (定量时间或时间额度, 即时间片), 23
- Timer interrupt (定时器中断), 492
- Time sharing (分时共享, 或简称分时), 218
- Time-sharing OS (分时操作系统), 31
- Time slice (时间片), 447
- Timestamp (时间戳), 400
- TLB (快表, 又称为转换后援缓冲器), 391
- TLB miss (快表的未命中率), 229
- Token passing bus (令牌传送式总线), 337
- Token Ring (令牌环网; 令牌环算法), 300, 308, 346, 348, 402, 404f, 457
- Top-half organization (上半部组织结构), 120
- Top level domain (TLD, 顶级域名), 343
- Topologies (拓扑结构), 335-338, 336-338f
- TopView (IBM 公司研发的一种多处理 8x86 环境), 417
- TORQUE (万亿级开源资源与队列管理器), 145
- Torvalds, Linus (李纳斯·托瓦兹), 114, 117
- Traceroute (一款用来发现连接两台网络主机之间的一系列路由器的实用例程, 即 Tracert), 354
- Tracert (一款用来发现连接两台网络主机之间的一系列路由器的实用例程), 354
- Track buffer (磁道缓冲), 323
- Tracking cookies (跟踪式小型文本文件), 366
- Tracks (磁道), 54, 302, 302-303
- Traffic monitor (通信监视器), 379
- Transaction (事务), 405, 432
- Transaction abort (事务夭折, 或称为事务中止), 406
- Transactional file system (事务型文件系统), 295
- Transaction commit (事务提交), 406
- Transaction processing (事务处理), 31, 295
- Transaction start (启动事务), 406
- Transaction support (事务支持), 432, 434
- Transfer (传输), 304-305, 305t
- Transient (瞬时的), 185, 188
- Transitions (状态转换), 154-156, 154f
- Translate (转换成), 210
- Translation lookaside buffer (TLB, 快表, 又称为转换后援缓冲器), 172, 228, 228f
- Transmission control protocol (TCP, 传输控制协议), 341
- Transparency, lack of (透明性, 缺乏), 387
- Transparent bridge (透明网桥), 347
- Transport layer (传输层), 334, 341-342
- Transport layer security (TLS, 传输层安全协议), 378
- Traps (异常, 或称为陷入事件), 495
- Trash (垃圾箱, 或称为回收站), 96
- Tree directory structure (树形目录结构), 260f
- Tree structure (树形结构), 260-261, 260f
- Trojans (特洛伊木马, 或简称木马), 361
- True identifiers (真正的标识符), 394
- TrueType (一种字体管理程序及字型标准), 102
- Trusted third party (TTP, 可信的第三方), 375
- Tunneling (网络隧道), 396
- Twisted pair (双绞线), 352
- Two disk drives (两个磁盘驱动器, 或简称双磁盘), 306f
- Two-factor authentication (双因素身份认证, 或称为双重身份认证), 367
- Two-level page table (两级页表), 231, 231f
- Two-phase commit (两阶段提交), 406

U

UCLinux (一种嵌入式 Linux 操作系统), 446

- UDF (基于统一光盘格式的文件系统类型), 428
 - UltraSPARC® (超级可伸缩处理器体系结构), 233
 - Unconditional jump operation (无条件跳转操作), 489f
 - Unicast (单点传送), 184, 186
 - Unicode Consortium (统一码联盟, 或称为统一码协会), 106, 107, 435
 - Unicode support (统一字符编码标准支持), 106-107
 - Unics (UNIX 操作系统名称的另一种写法). 见 UNIX
 - Uniform resource locator (URL, 统一资源定位符), 339
 - University of California San Diego (加利福尼亚大学圣地亚哥分校), 39
 - University of Helsinki (赫尔辛基大学), 114
 - University of Wisconsin (威斯康星大学), 136, 137
 - UNIX (UNIX 操作系统), 286-287, 460t
 - Unmarshalling (解组), 396
 - Unqualified name (非限定文件名), 263
 - Unreliable datagram (不可靠的数据报), 341
 - Unshielded twisted pair (UTP, 非屏蔽型双绞线), 352
 - USB (Universal Serial Bus, 通用串行总线), 106, 119, 438, 446
 - Use bit (使用位), 242
 - User account control (UAC, 用户账户控制), 419
 - User applications (用户应用程序), 74-75
 - User datagram protocol (UDP, 用户数据报协议), 340-341, 404, 474
 - User interface (UI, 用户界面), 24, 388, 478
 - User interface I/O devices (用户接口型输入/输出设备), 502
 - User-level thread (用户级线程), 169-170
 - User mode (用户模式, 也称为目态), 29, 490
 - User-mode driver framework (UMDF, 用户模式驱动程序框架), 419
 - User number (用户编号), 56
 - User optimization vs. hardware optimization (用户最优化与硬件最优化), 41
 - User OS environment (用户操作系统环境), 421-423
 - User release (“用户”版本), 116
 - Users of OS (操作系统的用户), 6-8, 7t
 - User view (用户视图), 6, 8
 - User-visible registers (用户可见寄存器), 493
 - Utilities (实用例程), 11
 - Utility programs (实用程序, 或称为实用例程), 344
 - Utilization (利用率), 132
- V
- Valid bits (有效位), 230, 242f
 - Variable number of processes (可变数量的进程), 218-221, 219f
 - VAX system (VAX 计算机系统), 184, 362, 440
 - VBScript (一种脚本语言), 365
 - VCalendar (日程安排交换格式标准), 474
 - VCard (电子名片交换格式标准), 474
 - Vector (向量), 400
 - V850 (NEC 公司的 v850 处理器), 446
 - Venter, J. Craig (J. 克雷格·文特尔), 143
 - Vertical distribution (垂直分布), 389-390
 - Vi command (一种文本编辑实用程序及命令), 264
 - Video (视频), 109
 - Video attributes (视频属性), 53
 - Video monitor output (视频监视器输出), 53-54
 - Vines (虚拟集成网络服务协议), 345
 - Virtual file system (VFS, 虚拟文件系统), 291, 291f, 292f, 452-454, 453f
 - Virtualizing (虚拟化), 38
 - Virtual machines (VM, 虚拟机), 38-40, 39f, 392-394, 392f, 393f
 - Virtual memory (VM, 虚拟内存, 或称为虚拟存储器), 101-103, 243, 425
 - Virtual organizations (虚拟组织), 140
 - Virtual page table (虚拟页表), 232
 - Viruses (病毒), 345
 - Virus scanners (病毒扫描器), 361
 - VisiCorp (VisiCorp 公司), 417
 - VisiOn (个人软件公司开发和发布的一款图形化界面支撑环境), 417
 - Vista (Windows Vista 操作系统), 432
 - Visual Basic (一种编程语言, 即 VB), 476
 - Volatile memory (易失性存储器), 498
 - Volume information (卷信息), 430
 - Volume layout (卷布局), 429f
 - Volume mount points (卷挂载点), 433
 - Volume name (卷名), 430

- Volume shadow copy (卷影复制), 432, 434
- Volume version (卷版本), 430
- Volunteer computing (志愿计算), 136-138
 - clusters (志愿计算集群), 144
 - systems (志愿计算系统), 130
- W
- Waiting (等待状态), 26
- Waiting semaphores (信号量申请操作), 193
- Wait routine (wait 例程), 193, 194
- Wait state (等待状态), 27, 155
- Wall time (挂钟时间), 131
- Warm standby (暖备份), 314
- Wavelength division multiplexing (WDM, 波分复用), 353
- Web interfaces (万维网界面, 或称为网站界面), 146-147
- Webserv (一个系统名称示例), 394
- Web servers (万维网服务器, 或称为网站服务器), 31, 484
- Well-known port number (标准端口的端口号), 339
- Well-known sockets (大家都熟悉的套接字), 189
- Wide area networks (WANs, 广域网), 335, 350-351, 506
- Wi-Fi (无线保真标准), 353, 473
- Window manager (窗口管理器), 459
- Windows (Windows 操作系统), 4
 - API (Windows 应用程序编程接口), 422
 - ME (Millennium Edition, 千禧版 Windows 操作系统), 417
 - Mobile (Windows 手机版操作系统), 68
 - 95 (Windows 95 操作系统), 417
 - 98 SE (second edition, 第二版的 Windows 98 操作系统), 417
 - NT 5.0 (Windows NT 5.0 版内核), 418
 - Server 2003 (Windows 2003 服务器版操作系统), 418
 - 6.0 (Windows 6.0 版内核), 418
 - 3.x (Windows 3.x 操作系统), 109, 421-422
 - 3.0 (Windows 3.0 操作系统), 417
 - 2000 (Windows 2000 操作系统), 33, 373, 418
 - Win32 (Windows 操作系统 Win32 库例程), 250
 - XP (Windows XP 操作系统), 250-251, 395, 418, 422
 - XP x64 Edition (64 位 Windows XP 操作系统), 418
- Windows NT (Windows NT 操作系统), 415-442
 - family architecture of (Windows NT 系列操作系统体系结构), 422f
 - family history of (Windows NT 系列操作系统发展例程), 416-421
- Windows on Windows Virtual DOS Machine (WOW VDM, 在 Windows 操作系统的虚拟 DOS 计算机上的窗口), 423
- WINS server (Windows 网络命名服务的服务器), 461
- Win32 (Win32 接口), 417
- Wireless access protocol (WAP, 无线访问协议), 479
- Wireless LANs (无线局域网), 308
- Wireless markup language (WML, 无线标记语言), 479
- Wireless networking (无线网络), 353
- Wire speed (线速), 338, 347
- Wiring concentrator (接线集中器, 又称为集线器), 346
- WordPerfect (一种文字处理程序), 92
- Words (字, 一种存储单位), 496
- Worker thread (工作线程), 392
- Workflows (工作流), 130-132, 130f, 131f, 137
- Working set (工作集), 240-242, 241t, 242f, 243f
- Work time (工作时间), 131
- WorldScript (一种为除了英语之外的语言提供系统级支持的程序, 寓意世界文字), 102
- World Wide Web (WWW, 万维网), 68, 110, 398, 474
- Worms (蠕虫), 362
- Worst fit (最坏适应算法), 219
- Wrap-up phase (程序的总结报告阶段), 215
- Write-back (写回结果), 494
- X
- XADD instruction (英特尔处理器的一条指令 XADD), 193
- Xerox Palo Alto Research Center (PARC, 施乐帕洛阿尔托研究中心), 90, 362, 417
- Xerox Star (施乐之星, 一款早期的计算机系统), 90
- X.500 (X.500 目录服务网络标准), 395

XFS (一种文件系统), 295

XNS (施乐网络系统协议), 184

XPBS (Globus 集群中间件调度程序的一种图形化用户界面版本), 146

X-Terminal (X 终端), 388

X-Windows (一种图形化用户界面), 417

X-Window (X11) system (X-Window 系统协议, 即 X11 系统协议), 458-459

Y

Y2K bug (千年虫问题), 41, 101

Z

Zilog Z-80 (美国齐格洛公司设计的 Z80 处理器), 50

Zombies (僵尸), 363

Zone bit recording (ZBR, 区位记录), 303

Z/VM (一款硬件级虚拟机, 即仿真软件包), 38